Herramienta de perfilado para código de bytes de Java

Proyecto de Sistemas Informáticos

Autores:

Carlos Loredo Iglesias
Sergio Ortiz Gil
Héctor Valles Mercado

Directoras:

Elvira Albert Albiol
Purificación Arenas Sánchez

Facultad de Informática.
Universidad Complutense de Madrid.
Curso 2009-2010.
Julio 2010.

RESUMEN

En este trabajo hemos implementado una herramienta para hacer "profiling" de programas escritos en código de bytes de Java, es decir, para contabilizar el número de recursos consumidos durante la ejecución de un programa. Los recursos que consideramos son: el número de instrucciones bytecode ejecutadas, el número de llamadas a un determinado método y el número de objetos creados (sin tener el cuenta el recolector de basura).

La herramienta o profiler recibe como entrada un fichero ".class" que contiene el código compilado de un programa Java. Junto con este fichero se debe especificar el nombre del método sobre el cual queremos hacer "profiling" y pasarle al interprete el valor concreto de los parámetros de entrada que se quieren utilizar para la ejecución. Una vez fijado el método y sus argumentos, el siguiente paso consiste en determinar qué recursos (de los ofrecidos por el sistema) se quieren medir. Con estos datos de entrada, en una primera fase el profiler transforma el ".class" a una representación intermedia denominada RBR ("Rule-Based recursive Representation). La RBR es un conjunto de reglas recursivas que contienen la información suficiente para poder simular, a partir de ellas, la ejecución del programa tal cual lo haría la máquina virtual de Java.

Para hacer el proceso de profiling más cómodo y sencillo para el usuario, hemos desarrollado una interfaz web en el lenguaje PHP que permite utilizar nuestro sistema de una manera muy intuitiva.

SUMMARY

In this work we have implemented a tool which allows to do profiling of Java bytecode programs, i.e., to count the number of resources required by the execution of a program. The resources of interest are: the number of executed bytecode instructions, the number of calls to a concrete method and the number of created objects without considering the garbage collector.

The profiler receives as input a ".class" file containing the compiled code of a Java program together with the name of the method for which the profiling will be done. Since we need to simulate the execution of the method at hand, the user must provide also the input arguments for the method. With this information, the profiler transforms the ".class" into an intermediate Ruled-based recursive Representation (RBR for short). The RBR is a set of recursive rules containing all the information required to execute the program as done by the Java Virtual Machine.

Furthermore, the profiler also supports a Web interface, developed in the language PHP, which allows the use of the system in a friendly way.

PALABRAS CLAVE

profiling, perfilado, java, bytecode, costa, coste, recursos, rbr

ÍNDICE

INTRODUCCIÓN

El proyecto de Sistemas Informáticos recogido en esta memoria se centra en el análisis de los recursos que un algoritmo necesita para completar correctamente su ejecución o para que un determinado problema sea resuelto con éxito, en función de los datos o argumentos de entrada. Se engloba, por tanto, dentro de la teoría sobre complejidad computacional.

Concrétamente, en lugar de realizar el análisis sobre algoritmos o resolución de problemas, hemos optado por el estudio de programas, esto es, implementaciones específicas de algoritmos concretos para la resolución de problemas determinados.

La principal motivación de este proyecto es poder obtener de manera automática y sin intervención humana los recursos que un determinado programa necesita durante el intervalo en el que se ejecuta para que éste realice su tarea de manera correcta.

Algunas áreas donde es interesante la aplicación de este proyecto son:

- Determinación de los límites de uso de los recursos: este tipo de análisis puede servir para certificar que los recursos proporcionados por un entorno concreto son suficientes para la ejecución de un determinado programa de manera correcta.
- Optimización del rendimiento: mediante este tipo de análisis se pueden delimitar las partes en las que un programa hace uso intensivo de determinados recursos y, a partir de ahí, centrarse en optimizar esas partes para que el rendimiento global aumente.

Tradicionalmente, para realizar el análisis en el que este proyecto se centra, se ha venido utilizando el código fuente de los programas a tratar. En nuestro caso concreto hemos decidido utilizar el código objeto en lugar de éste primero. El motivo de ello es que en programas comerciales o de terceros no suele presentarse el código fuente y son éstos en los que más interesa realizar el tipo de análisis mencionado. Un ejemplo de esto son las aplicaciones Java para móviles que se descargan desde internet y de las que solo es posible tener acceso al codigo de bytes.

Específicamente, el código objeto que se va a utilizar es código de bytes o bytecode generado a partir de código fuente escrito en el lenguaje de programación Java, aunque no se debería limitar solo a este lenguaje de programación sino a cualquiera que pueda ser transformado a código de bytes compatible. Más concretamente a cualquiera que pueda ser traduducido a una representación intermedia comentada más adelante que será lo que realmente nuestro proyecto interprete y ejecute.

Una muy buena característica del código de bytes que vamos a utilizar para realizar el análisis es que es necesario el uso de una máquina virtual para su ejecución y esto hace que dicho código de bytes sea independiente de la plataforma donde se va a ejecutar. Gracias a esto el análisis

realizado sobre un único código de bytes puede aplicarse a todas las diferentes plataformas donde una máquina virtual para éste esté disponible.

Por otro lado, la aproximación elegida para realizar el análisis es eminentemente práctica y va a consistir en ejecutar el código de bytes proporcionado e ir contabilizando los recursos que son necesarios para llevar a cabo dicha ejecución de principio a fin.

De todos los posibles recursos a tener en cuenta, nuestro sistema será capaz de contabilizar el número de instrucciones ejecutadas, el número de llamadas que se realizan a métodos definidos en el código de bytes y el número de objetos creados durante la ejecución.

Con esta información será posible trabajar en las áreas de mayor aplicación citadas anteriormente:

- el número de instrucciones ejecutadas y el número de objetos creados delimitarán la complejidad en tiempo y en espacio, respectivamente, que la ejecución del programa ha presentado.
- el número de llámadas a métodos permitirá concentrarse en las zonas del código de las que se ha hecho uso intensivo para intentar optimizarlas y mejorar el rendimiento global.

Para llevar a cabo la ejecución de código de bytes es necesario un intérprete que sea capaz de entender y procesar cada una de las instrucciones contenidas en éste y, a su vez, mantener cada uno de los resultados intermedios que dichas instrucciones producen y que son necesarios para que continúe la ejecución correctamente; esto es, el estado de la máquina. Por ello es necesario implementar una máquina virtual del código de bytes.

Diseñar e implementar desde cero una máquina virtual completa que permitiera el ańalisis del que venimos hablando junto con el análisis propiamente dicho hubiese sido una tarea demasiado ambiciosa para la asignatura de Sistemas Informáticos y es por ello la razón por la que nuestro sistema se basa y apoya en otro cuyo nombre es COSTA[1].

El sistema COSTA, COSt and Termination Analyzer for Java bytecode, trata de obtener los límites superiores del uso que un código de bytes hace de los recursos a analizar mientras que nuestro proyecto medirá los recursos utilizados exactamente durante la ejecución. La aproximación que realiza el sistema COSTA es una ejecución estática y los límites superiores obtenidos mediante este método siempre estarán por encima de los resultados obtenidos por nuestro proyecto.

Una de las tareas que realiza el sistema COSTA para poder realizar los análisis pertinentes es transformar el código de bytes inicial en una representación intermedia equivalente denomidada rule-based recursive representation o RBR. Mediante ésta el sistema COSTA es capaz de inferir el ánalisis citado sin necesidad de la ejecución del código de bytes.

1 ELVIRA ALBERT, PURI ARENAS and SAMIR GENAIM, Cost Analysis of Object-Oriented Bytecode Programs

La principal ventaja de esta representación intermedia es que es de más alto nivel que el código de bytes y esto hace que su interpretación y ejecución sea una tarea mucho más sencilla. Junto a esto, en ella ya viene resuelta la invocación virtual y el manejo de excepciones. Además, la estructura mediante la cual está definida hace que, para proceder a su ejecución, un lenguaje como Prolog pueda servirse de los mecanismos de vuelta atrás o backtraking que éste último lleva incorporados. Todo esto hace del proceso de interpretación una tarea mucho más sencilla.

Estas son las razones por las que el lenguaje que hemos elegido para implementar el intérprete o máquina virtual sea Prolog y ello ha hecho que el código resultante sea sencillo, de fácil mantenimiento y no excesivamente largo.

Finalmente, cabe decir que tanto los resultados proporcionados por el sistema COSTA como por los proporcionados por el sistema comentado en esta memoria son complementarios y esto hace que sea posible el contraste entre ambos para afianzar los análisis.

JAVA BYTECODE

Introducción a Java

En 1991 comenzó el proyecto del lenguaje Java. En un primer momento se llamó OAK[2] y fue ideado pensando que se usaría para programar los contenidos de la televisión interactiva. La idea fue desestimada debido a su alta complejidad. Durante ese tiempo, además, se barajaron varios nombres hasta acabar llamándose Java.

En 1995 se lanzó la primera versión, Java 1.0, que utilizaba un lenguaje de programación con una notación similar a C y prometía que los programadores sólo tendrían que programar una aplicación que funcionaría en cualquier plataforma.

La máquina virtual

Para abstraerse de la arquitectura de la máquina que ejecuta el código es necesario disponer de un entorno virtual cuyas instrucciones se adapten a las del computador y sistema operativo correspondiente. Esto es lo que se conoce como máquina virtual.

La máquina virtual es -como muchas máquinas físicas y virtuales- una máquina basada en una arquitectura de pila afín a un microcontrolador (procesador), provista de marcos que constituyen ámbitos, donde se alojan resultados parciales, parámetros, valores de retorno de funciones, excepciones, y que sirven, además, para llevar a cabo el enlazado dinámico. La máquina virtual de Java no acepta los tipos dinámicos sino que éstos son fijos. Esto obliga que Java sea un lenguaje fuertemente tipado.

La máquina virtual utiliza variables de pila, variables locales (a nivel de método), *fields* (a nivel de clase), referencias al heap y constantes que toma de la *constant pool*[3].

Para liberar aún más al programador de tareas no relacionadas con la algoritmia, como es, la gestión de memoria, la máquina virtual gestiona la memoria por sí misma. Esto hace que no se puedan referenciar regiones de memoria utilizando punteros, como en los lenguajes de programación clásicos. Hay que tener en cuenta que las referencias a los objetos (atributos) sólo son posibles dentro del ámbito en el que están declarados. Fuera de él, el acceso a los datos correctos no está garantizado -y la máquina virtual devolverá siempre *null* en ese caso- ya que la máquina virtual los marca para ser eliminados en la próxima recolección de basura que, además, puede llevarse a cabo bajo demanda o cuando se detecta que se está acabando el espacio de forma automática. En ese caso se desconoce cuándo se va a llevar a cabo aunque se minimizará la recolección para no penalizar continuamente el rendimiento.

2 WIKIPEDIA, Java (programming language), History, http://en.wikipedia.org/wiki/Java_(programming_language)

3 TIM LINDHOLM and FRANK YELLIN, The Java™ Virtual Machine Specification, Second Edition, Chapter 3: The structure of the Virtual Machine, Runtime Constant Pool

La gestión de memoria de la máquina virtual garantiza que no se queda ocupada ninguna dirección de memoria de forma irrecuperable tras la creación de un objeto. Así pues, los llamados "memory leaks" dejan de existir. Por supuesto, el programador es el responsable de que los datos dejen de ser referenciables -cambiando de ámbito- cuando ya no vayan a ser usados para contribuir a la desocupación de la memoria.

La máquina virtual interpreta -las primeras implementaciones- o compila y ejecuta -para mejorar el rendimiento- archivos binarios .class cuyas instrucciones son bytecodes. Antes de este proceso, la máquina virtual realiza tres verificaciones antes de comenzar la ejecución, comprobando:

1. Que los saltos son a regiones de código y que su destino está dentro del mismo método.
2. Que los datos siempre son inicializados. En concreto, los tipos básicos a cero y los objetos a *null*.
3. Que se respetan todas las restricciones establecidas por los modificadores de acceso: *private*, *protected* y *package*.

La máquina virtual, además de utilizar las bibliotecas predefinidas de Java correspondientes a su versión, ha de implementar de forma nativa algunos métodos, es decir, métodos que son dependientes de la plataforma sobre la que esté ejecutándose la máquina. Por ejemplo, los métodos de acceso a la pila de llamadas para el tratamiento de excepciones y los métodos que en última instancia llevan a cabo operaciones de entrada salida son nativos.

El ancho de palabra depende de la plataforma que implemente la máquina virtual. Así pues, si se ejecuta en una máquina un programa de 64 bits, el tamaño de palabra será de 64bits y los tipos que habitualmente reservan una palabra desperdiciarán la mitad de los datos en esta implementación nativa, sin embargo, los tipos que ocupan 64bits (habitualmente doble palabra) cabrán en una sola y no desperdiciarán espacio.

El formato binario compatible con la máquina virtual

La máquina virtual utiliza archivos binarios a modo de ejecutables como puede hacerlo un sistema operativo. De hecho, pueden compilarse otros lenguajes[4], como C, ADA,COBOL o PYTHON entre otros, utilizando el repertorio de instrucciones (bytecode) de la máquina virtual y se pueden ejecutar si pasan las tres comprobaciones que realiza la máquina virtual.

Un archivo .class contiene la siguiente estructura[5]:

```
ClassFile {
   u4 magic;
   u2 minor_version;
   u2 major_version;
```

4 WIKIPEDIA, Java virtual machine, JVM languages http://en.wikipedia.org/wiki/Java_virtual_machine

5 TIM LINDHOLM and FRANK YELLIN, The Java™ Virtual Machine Specification, Second Edition, Chapter 4: The class File Format, The ClassFile Structure

```
u2 constant_pool_count;
cp_info constant_pool[constant_pool_count-1];
u2 access_flags;
u2 this_class;
u2 super_class;
u2 interfaces_count;
u2 interfaces[interfaces_count];
u2 fields_count;
field_info fields[fields_count];
u2 methods_count;
method_info methods[methods_count];
u2 attributes_count;
attribute_info attributes[attributes_count];}
```

Descripción de los campos:

- *magic*: debe ser `0xCAFEBABE` para asegurarse de que es un achivo .class.
- *minor_version* y *major_version*: versión de la máquina virtual para la que se compiló la clase o interfaz.
- *constant_pool*: contiene las contantes, en concreto *constant_pool_count* es la cantidad de constantes que contiene.
- *access_flags*: son los marcadores de permisos de acceso de la clase tales como *public*, *private*, *final*, *super* e *interface*.
- *this_class*: índice en la *constant_pool* donde se guarda información de la clase o interfaz.
- *super_class*: la clase de la que hereda directamente. La única que no hereda de ninguna otra es *Object*.
- interfaces: contiene los interfaces que implementa. En concreto, el número de interfaces es *interfaces_count*.
- *fields*: contiene información sobre los atributos de la clase.
- *atributes*: son los atributos. Éstos también se encuentran en la información de los métodos y de las clases.

Uno de los atributos dentro de *method_info*[6] (hay que recordar que hay uno por método) es el más importante para nuestro proyecto ya que contiene un atributo de tipo *code_attribute*:

```
Code_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 max_stack;
  u2 max_locals;
  u4 code_length;
  u1 code[code_length];
  u2 exception_table_length;
  {
    u2 start_pc;
    u2 end_pc;
    u2 handler_pc;
    u2 catch_type;
  }
```

6 TIM LINDHOLM and FRANK YELLIN, The Java™ Virtual Machine Specification, Second Edition, Chapter 4: The class File Format, Attributes , The Code Attribute

```
    exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Entre otros campos de la estructura, aquí se encuentra la mayor profundidad que puede alcanzar la pila en el método, el máximo número de variables locales que habrá y una tabla de excepciones.

Es dentro de *code* donde se encuentran los bytecodes, instrucciones que la máquina virtual es capaz de ejecutar. En cada método hay una cantidad de bytecodes definida en *code_length*.

El Bytecode, la instrucción de la máquina virtual de Java

Las instrucciones de la máquina virtual se conocen con el nombre de bytecodes debido a que el código de una operación ocupa un byte. Por tanto, existen 256 bytecodes diferentes.

Para no dejar incompleta esta memoria y al mismo tiempo, no alargar esta sección, ahora mostraremos los *simple bytecodes*[7] de Java. Al final de dicha memoria se incluye un anexo donde se encuentran todos los bytecodes que debe ejecutar la máquina virtual y su formato.

Instrucciones que manipulan operandos en la pila:

- load X: apilan la variable local X.
- store X: guardan la cima de la pila en la variable local X.
- push X: apilan operandos constantes explícitos.
- pop: desapilan elementos sin utilizarlos.
- Dup: duplican la cima de la pila convenientemente.

Instrucciones aritmetico-lógicas:

Todas ellas toman operandos de la pila y alojan en la misma su resultado:

- add: suma dos operandos.
- mul: multiplica dos operandos.
- sub: resta dos operandos.
- div: divide dos operandos.

Se dispone también de las habituales operaciones lógicas binarias:

- and, or, xor.

Y de una unaria:

- not.

7 ELVIRA ALBERT, PURI ARENAS and SAMIR GENAIM, Cost Analysis of Object-Oriented Bytecode Programs, página 5.

Instrucciones para llevar a cabo el control del flujo de ejecución:

- ifXX D: desapilan, comparan con 0 un operando de la pila y, si se satisface, la condición saltan a D.
    - En referencias existe ifnull que saltará si la referencia es null.
- goto D: salta incondicionalmente a la dirección D.

Manipulación de objetos:

- new I: crea un objeto del tipo I.
- getfield I: obtiene el valor del atributo I.
- putfield I: guarda el valor en un atributo I.

Manipulación de arrays:

- newarray T: crea un array de tipo T.
- aload: permite apilar desde la pila a una posición de un array.
- astore: guarda en la posición de un array un elemento que desapila.
- arraylength, permite conocer la longitud de un array

Gestión de métodos:

- invokevirtual M: llama al método virtual M.
- invokenonvirtual M: llama al método no virtual M.
- return: regresa del método actual.

Ejemplo de traducción a simple bytecode de cada una de sus instrucciones Java[8]:

Se muestra a la izquierda el código Java, los *simple bytecodes* en el centro y a la derecha lo que hará la máquina virtual durante la ejecución de los mismos.

```
class A {
   int incr(int i) {
      return i+1;
   }
}

class B extends A {
   int incr(int i) {
      return i+2;
   }
}

class C extends B {
   int incr(int i) {
      return i+3;
   }
}

class Main {
   int add(int n,A o) {
      int res=0;
      int i=0;
      while (i<=n) {
         res=res+i;
         i=o.incr(i);
      }
   return res;
   }
}
```

8 Este ejemplo ha sido tomado del artículo:
ELVIRA ALBERT, PURI ARENAS and SAMIR GENAIM, Cost Analysis of Object-Oriented Bytecode Programs, página 6.

int incr(int i) {		
return i+1;	1: load 1	Se apila la primera variable local.
	2: push 1	Se carga la constante 1.
	3: add	Se suman y se apila el resultado.
}	4: return	Regresa del método.
}		

---

class B extends A {		
int incr(int i) {		
return i+2;	1: load 1	Se apila la primera variable local.
	2: push 2	Apila la constante 2.
	3: add	Se suman y se apila el resultado.
}	4: return	Regresa del método.
}		

---

class C extends B {		
int incr(int i) {		
return i+3;	1: load 1	Se apila la primera variable local.
	2: push 3	Apila la constante 3.
	3: add	Se suman y se apila el resultado.
}	4: return	Regresa del método.
}		

---

class Main {		
int add(int n,A o) {		
int res=0;	1: push 0	Apila la constante 0.
	2: store 3	Guarda en la tercera variable local.
int i=0;	3: push 0	Apila la constante 0.
	4: store 4	Guarda en la cuarta variable local.
while (i<=n) {	5: load 4	Se apila la cuarta variable local.
	6: load 1	Se apila la primera variable local.
	7: ifgt 17	Salta a 17 si mayor que 0.
res=res+i;	8: load 3	Se apila la tercera variable local.
	9: load 4	Se apila la cuarta variable local.
	10: add	Se suman y se apila el resultado.
	11: store 3	Guarda en la tercera variable local
i=o.incr(i);	12: load 2	Se apila la segunda variable local.
	13: load 4	Se apila la cuarta variable local.
	14: invokevirtual A.incr:(I)I	Llamada a A.incr, pasa y devuelve int
	15: store 4	Guarda en la cuarta variable local..
}	16: goto 5	Salto a la dirección 5.
return res;	17: load 3	Se apila la tercera variable local.
}	18: return	Regresa del método.
}		

REPRESENTACIÓN RECURSIVA BASADA EN REGLAS (RBR)

El sistema COSTA, que explicaremos más adelante, traduce las instrucciones de Java (bytecodes) en un formato más adecuado para poder trabajar desde PROLOG , partiendo siempre desde un archivo con extensión *.class* de java. Este formato es el que llamaremos representación recursiva basada en reglas o RBR[9].

A lo largo de esta sección, utilizaremos las palabras regla y bloque indistintamente puesto que se puede considerar lo mismo en la RBR.

Fundamentos de la representación RBR desde el punto de vista de la ejecución

1. Una RBR se compone de un conjunto finito de reglas o bloques. De todas ellas, una es la que inicia la ejecución y el resto las que realizan la ejecución en sí misma.
2. Toda regla pertenece a un único método definido en el archivo *.class* de Java.
3. Las reglas pueden contener instrucciones (bytecodes), guardas o llamadas a otras reglas.
4. El paso de una regla a otra puede ser directo o, en caso de que existan varias alternativas, regulado mediante aserciones o guardas.
5. La recursión es la única forma de iterar entre reglas.
6. Sólo existen variables locales. La pila se aplana, quedando sustituida por un tipo de variables locales dedicadas a aportar la misma funcionalidad coherentemente.
7. Para el manejo de objetos se requiere un Heap.
8. Algunas de las características de los objetos no están disponibles:
    1. Los objetos son registros con un campo que contiene su tipo.
    2. El comportamiento de la asignación dinámica de métodos (métodos virtuales) se contempla creando un bloque para cada posible llamada.
9. No existe diferencia entre ejecutar un bloque y un método, de hecho, los métodos son uno o varios bloques.

La estructura de reglas se ordena utilizando el nombre del método al que pertenece si ésta es la primera regla del método o mediante un índice numérico diferente por cada regla posterior. Sin realizar un grafo de dependencias no se puede saber a qué método pertenece una regla con índice numérico y, por tanto, únicamente mediante las guardas se puede dirigir la ejecución una vez que ésta se inicia en la primera regla. Todas las reglas que genera el sistema COSTA deben ser accesibles en cualquier momento.

En caso de que haya saltos, durante la generación de la RBR, se crea una regla nueva siempre que haya uno tanto condicional como incondicional. Esto es debido a que las instrucciones de cada regla siempre se ejecutan secuencialmente.

Las guardas que se utilizan son siempre disjuntas. Por tanto, para interpretar la ejecución en PROLOG, basta con aprovechar la unificación con la regla adecuada en función del estado de la ejecución y el backtracking. Al ser las guardas disjuntas, el indeterminismo nunca es un problema.

9 ELVIRA ALBERT, PURI ARENAS and SAMIR GENAIM, Cost Analysis of Object-Oriented Bytecode Programs, página 10.

Cuando se ha finalizado la ejecución del último bloque se puede considerar que el programa ha terminado.

Además, siempre se generan reglas para el tratamiento de todas las posibles excepciones. Cuando el programa finaliza es necesario saber qué reglas se han escogido para saber si éste terminó de manera anómala (excepción no capturada) o de manera normal.

Estructura de la RBR

Todo bloque posee una cabecera, seguida de un operador := y un conjunto de instrucciones como cuerpo. Éstas pueden ser:

1. Bytecodes traducidos directamente de Java cuya semántica no varía. Se contemplan al hacer el profiling y al realizar la ejecución.
2. Aserciones o guardas, que son necesarias para comprobar condiciones y realizar correctamente el paso de un bloque a otro. No se contemplan al hacer el profiling pero sí al realizar la ejecución.
3. Instrucciones de tipo *call* que sustituyen a las invocaciones y a los saltos condicionales y no condicionales. No se contemplan al hacer el profiling pero sí para realizar la ejecución.
4. Nops o instrucciones nulas que no se utilizan para realizar la ejecución pero que se pueden tener en cuenta a la hora de realizar el profiling. Pueden ser:
    1. Instrucciones de invocación a métodos que tienen una marca para saber que se debe considerar su coste.
    2. Saltos condicionales y no condicionales que tienen una marca para saber que se debe considerar su coste.
    3. Instrucciones que dependen de un resultado condicional anterior que tienen una marca para saber que no se debe considerar su coste.

Variables

En los bloques se puede hacer uso de tres tipos de variables[10]:

1. Las variables locales que, a diferencia de java, son locales a bloque y no a cada método. Su acceso es directo y se representan como l(n), donde n es un número natural.
2. Las variables de pila, que representan la pila aplanada y cuyo acceso es directo al igual que en las variables locales. Sólo se diferencian de ellas para hacer más comprensible el funcionamiento y para que la traducción de código sea más natural. Así, si un bytecode toma operandos de la pila, la RBR utilizará variables de pila representadas como s(n), donde n es un número natural.
3. Las variables de excepción, que se utilizan como variables locales y guardan referencias a objetos creados por una excepción. Es posible que estas variables no almacenen ninguna

10 ELVIRA ALBERT, PURI ARENAS and SAMIR GENAIM, Cost Analysis of Object-Oriented Bytecode Programs, página 12.

referencia durante parte o toda la ejecución, pero, es necesario propagarlas para guardar las excepciones por si alguna de éstas se produce.

Es indispensable que en la implementación exista un Heap donde se guarden los objetos a los que variables locales, de pila y de excepción puedan hacer referencia.

La cabecera de una regla

Es el primer elemento de la regla y está compuesta de:

- Un identificador, que puede ser[11]:
  - Una cadena de caracteres del tipo *nombre-clase_nombre-del-método(parámetros-del-método -en-java)tipo-devuelto-en-Java,* si el bloque es el primero de un método. Esta cadena es única para el método.
  - Un número y además ir seguido de texto si pertenece a un bucle. Estos identificadores comienzan siendo 1 y este número se va incrementando a medida que se crean nuevas reglas de ése u otros métodos.

  En el segundo caso es posible que haya más de una regla con el mismo identificador, lo que supone que para llegar a ella hay que tomar una decisión sobre cuál elegir. La manera de elegir una concreta es mediante las aserciones o guardas.

- A continuación y entre paréntesis, la lista de parámetros de entrada, la lista de parámetros de salida, la lista de las excepciones y la lista que a modo de grafo de dependencia implementa el estado de la máquina. Este último puede corresponderse con *normal* o *exception* en función de lo ocurrido durante la ejecución.

Ejemplo de la cabecera de una regla para el método main inicial estándar de Java, que debe ser del tipo *static void main(String args[])*:

paquete/clase_main([Ljava/lang/String;)V([l(0)],[],[e(1)],[A])

Donde:
- [Ljava/lang/String; identifica al tipo: String[] .
- l(0) es el nombre local de una variable de entrada.
- la siguiente lista de parámetros de salida está vacía ya que el método es void.
- hay una variable de salida donde se alojaría una hipotética excepción.
- A es igual que una variable Prolog y al acabar de ejecutar habrá quedado unificada con '*normal*' o *'exception'*, así sabremos el resultado al acabar.

Ejemplo de la cabecera de un bloque intermedio cualquiera, podría ser:

8([l(1), l(2), s(1)],[l(1), l(2)],[l(1), l(2), e(1)],[A])

Donde:
- el bloque está identificado con 8.

11 ELVIRA ALBERT, PURI ARENAS and SAMIR GENAIM, Cost Analysis of Object-Oriented Bytecode Programs, página 14.

- [l(1), l(2), s(1)] es la lista de las variables de entrada. La última de ellas es de pila.
- la siguiente lista [l(1), l(2)], contiene los nombres locales de las variables de salida.
- hay tres una variables de salida [l(1), l(2), e(1)] la última de ella podría contener excepciones.
- A es igual que una variable prolog y al acabar de ejecutar habrá quedado unificada con 'normal' o 'exception', así sabremos el resultado al acabar.

Ejemplo de cabecera de bloques que genera un bucle:

Al que se salta:
2_loop([l(1), l(2)],[l(1), l(2)],[l(1), l(2), e(1)],[A])

En el que acaba:
2_loop_nexit([l(1), l(2)],[l(1), l(2)],[],[normal])

Nótese que en el último bloque la variable de tipo Prolog que lleva el control de la ejecución queda instanciada a *normal*, el resto son idénticas a las vistas enteriormente.

Paso y retorno de variables. El bytecode call.

Cuando se llega a un bytecode call, éste tiene indicado cuál es el bloque siguiente, una lista con los parámetros de entrada y otra con los parámetros de salida. El tamaño de éstas últimas exactamente se corresponderá con el tamaño de las listas de parámetros de entrada y de salida de la cabecera del bloque al que se va a saltar, respectivamente.

Es necesario que las variables locales, de pila y de excepción del bloque que contiene el bytecode call (a partir de ahora bloque anterior) sean renombradas y copiadas al bloque siguiente. Este renombramiento hará que cada n-ésima variable del bloque anterior que esté en la lista de variables de entrada del bytecode call cambie su nombre por el de la variable n-ésima de la lista de variables de entrada del bloque siguiente.

Toda variable que no se copie, será una variable independiente en el bloque siguiente.

Cuando la ejecución del bloque siguiente termina y se vuelve al bloque anterior, las variables de salida del bloque se renombran y copian, pero, esta vez, con los nombres de las variables que presenta la lista de variables de salida del bytecode call.

Además de las variables, es necesario que el estado de la máquina (lista de estados a modo de grafo de dependencia) se propague coherentemente.

Cabecera del método main estádar de Java:
paquete/clase_main([Ljava/lang/String;)V([l(0)],[],[e(1)],[A])
Llamada al método main de la clase *clase*, perteneciente al paquete *paquete*:
call(method(static), paquete/clase_main([Ljava/lang/String;)V, [[s(1)], [], [e(2)], [A]], [])

Llamada al bloque 8:
call(block, 8, [[l(1), l(2), s(1)], [l(1), l(2)], [l(1), l(2), e(1)], [A]], [])

Llamada de iteración del bucle visto arriba:
call(block, 2_loop, [[l(1), l(2)], [l(1), l(2)], [l(1), l(2), e(1)], [A]], [])

Llamada de salida del bucle:
call(block, 2_loop_nexit, [[l(1), l(2)], [l(1), l(2)], [], [A]], [])

Se puede ver que las variables no tienen por qué ser la mismas. En la llamada a main, la variable local de pila *s(1)* pasará a ser l(o) dentro del bloque de main.
El primer parámetro del bytecode call siempre indica si la llamada es a un bloque o un método. En este último caso además se informa de su tipo entre paréntesis.

Intrucciones Assr. Definición de guardas.

Para dirigir la ejecución, es necesario que se lleven a cabo dos comprobaciones.

La primera, que las variables que tiene el bytecode call para manejar el estado de la máquina deben poder unificarse con las de las reglas. Si en el ejemplo anterior, al hacer la llamada al bloque ***2_loop_nexit,*** la variable Prolog A estuviera unificada con *exception*, nunca se llamaría al bloque ***2_loop_nexit([l(1), l(2)],[l(1), l(2)],[],[normal])***, puesto que la variable ya unificada con *'exception'* no puede unificar con *'normal'. D*e esta forma se establecen las guardas para la gestión de excepciones.

La otra forma de gestionar guardas es mediante el uso de instrucciones assr, también llamadas aserciones. Éstas utilizan ciertas variables locales o de pila y hacen operaciones con ellas. Sólo cuando el resultado es el esperado tienen éxito y se puede continuar la ejecución tras la aserción. En caso contrario se debe volver hacia atrás y tomar la siguiente regla posible teniendo en cuenta de dónde se viene y el estado de la máquina. En estos casos el comportamiento es como la vuelta atrás o backtracking.

Que las aserciones sean disjuntas asegura que no exista ambigüedad.

Su sintaxis es la siguiente:

```
assr(tipo-de-aserción, argumentos-de-entrada, argumentos-de-salida)
```

Aunque aparece una lista de argumentos de salida, realmente no se utiliza ya que el único cambio posible en el estado de la máquina que pueden provocar es eliminar las variables locales utilizadas para realizar la comprobación.

Repertorio de instruciones assr y funciones que realizan

En esta sección veremos cuándo una instrucción assr es cierta, en caso contrario, siempre es falsa.

- Comparación con cero:
    - No Destructiva (no elimina el operando de entrada):
        - int_nonzero, cierto si el argumento de entrada es distinto de cero.
        - int_zero, cierto si el argumento de entrada es cero.

    - Destructiva (elimina el operando de entrada, si la condición se satisface):

        - cmpz(ne), cierto si no es igual que cero.
        - cmpz(eq), cierto si es igual que cero.
        - cmpz(gt), cierto si es mayor que que cero.
        - cmpz(ge), cierto si es mayor o igual que cero.
        - cmpz(lt), cierto si es menor que que cero.
        - cmpz(le), cierto si es menor o igual que cero.

- Comparación entre dos argumentos (elimina operandos de entrada, si la condición se satisface):
    - cmp(icmpeq), cierto si ambos argumentos de entrada son iguales.
    - cmp(icmpne), cierto si los argumentos de entrada son diferentes.
    - cmp(icmpge), cierto si el primer argumento es mayor o igual que el segundo.
    - cmp(icmpgt), cierto si el primer argumento es mayor que el segundo.
    - cmp(icmple), cierto si el primer argumento es menoor o igual que el segundo.
    - cmp(icmplt), cierto si el primer argumento es menor que el segundo.

- Comparación de referencias:
    - cmp(acmpeq), cierto si sus dos argumentos de entrada son referencias al mismo objeto (referencias iguales).
    - cmp(acmpne), cierto si sus dos argumentos de entrada son referencias diferentes.

- Casting:
    - can_be_casted(*Tipo*), cierto si el objeto referenciado en el argumento de entrada permite hacer un casting con el tipo indicado en *Tipo*.
    - cannot_be_casted(*Tipo*),cierto si el objeto referenciado en el argumento de entrada no permite hacer un *casting* con el tipo indicado en *Tipo* .

- Comparación con null:
    - cmpnull(nonnull) y nonnull, ambas ciertas si el argumento de entrada no es null.

    - cmpnull(null) y null, ambas ciertas si el argumento de entrada es null.

- Comparación de tipos:
    - type([*Tipo*]), cierto si el argumento de entrada es una referencia a un objeto del tipo indicado por*Tipo.*

- Instancia:
    - instanceof([*Tipos*]), cierto si el argumento de entrada es una referencia a un objeto que es instancia de todos los tipos de la lista *Tipos*.
    - not_instanceof([*Tipos*]), cierto si el argumento de entrada es una referencia a un objeto que no es instancia de ninguno de los tipos de la lista *Tipos*.

- Tratamiendo de arrays:
    - non_negative_array_size, Cierto si la referencia pasada es a un array con tamaño positivo.

- negative_array_size, Cierto si la referencia pasada es a un array con tamaño negativo.
- non_negative_multiarray_size, Cierto si recorriendo las referencias de cada subvector todos son de tamaño negativo.
- array_index_in_boundary, Cierto si el segundo argumento es un índice dentro del array al que apunta la referencia que es el primer argumento.
- array_index_out_of_boundary, Cierto si el segundo argumento es un índice fuera del array al que apunta la referencia que es el primer argumento.
- compatible_array_and_value_types, Cierto si la referencia que se encuentra primer argumento es a un array que es del mismo tipo que el objeto cuya referencia es el segundo argumento.

Los bytecodes de la representación RBR

Algunas de las operaciones que hemos visto anteriormente en los assr se realizan también con bytecodes. La diferencia está en si lo importante es discriminar una regla de otra (a modo de guarda) o disponer del resultado de una operación, por ejemplo de comparación, para utilizarla después.

Sintaxis de los bytecodes

En negrita vemos los parámetros de los bytecodes que hemos utilizado:
bytecode(dirección -*ignorado*-, **Identificador del bytecode**, **Parámetros de entrada**, **Parámetros de salida,** Estado de la Pila antes y después y próxima dirección -*ignorado*-).

Tipos de bytecodes, dependiendo de su identificador

- De retorno: return(tipo devuelto), siguiendo la estructura de bloques no es necesario para nuestra implementación.
- De carga en la pila:
  - aload(X), donde se carga desde el array que está alojado donde indica el primer argumento, la componente indicada por el segundo, X puede ser l, en cuyo caso es un tipo long y d. Lo guarda en las variables indicadas por la salida.
  - load(Tipo, longitud), carga utilizando el lugar de la memoria que se le pasa con un argumento y lo guarda en la variable que está como parámetro de salida.

- De guardado en memoria:
  - astore(X), donde se guarda en el array que está alojado donde indica el primer argumento, en la componente indicada por el segundo, el valor indicado por el tercer argumento, X puede ser l, en cuyo caso es un tipo long y d.
  - store(Tipo, longitud), Carga utilizando el lugar de la pila que se le pasa con un argumento y lo guarda en la variable que está como parámetro de salida.

- Operaciones binarias aritméticas: del tipo barithm(*Tipo, Operación*)
  - Llevan a cabo, dependiendo de *Operación* las operaciones add, mul, sub, div, rem, and , or, xor, not (unaria)...

    - *Tipo* identifica el tipo primitivo al que se refieren, i - enteros, f - float, d - doubles...
    - Toman una variable de entrada, aplican el operador a la segunda y alojan el resultado en la de salida, exceptuando not, que sólo lo hace con una, ya que es una operación unaria.

- Incremento: iinc(_, *Valor*): Suma Valor a la variable de entrada y devuelve el resultado en la variable de salida.

- Desplazamiento de bits:
    - XshY, toma un primer argumento como entrada y a él le desplazan tantos bits como indique el segundo argumento, alojando el resultado en la variable de salida, donde X es el tipo, *i, l* o *iu* ,e Y es *r* si es hacia la derecha o *l* si es hacia la izquierda.

- Pila:
    - push(Identificador de operación , tipo a apilar(*Valor*)), carga valor en la variable de salida, con el tipo adecuado.
    - push(Identificador de operación , *Valor*), carga valor en la variable de salida.
    - aconst_null, carga null en la variable de salida.
    - dup, dup_x1, duplican en la pila el contenido de la variable de entrada.
    - pop y pop2, eliminan dos variables de pila (el efecto de desapilar)
    - swap, intercambia dos variables de pila.

- Comparacíon:
    - cmpr(*Tipo*, _), toma dos variables del tipo *Tipo* como argumentos y las compara, si la primera es menor que la segunda la variable de salida valdrá -1, 0 si son iguales y 1 si la primera es mayor.

- Creación:
    - new(*Tipo*), reserva espacio para un objeto de tipo *Tipo* y devuelve el lugar en el Heap.
    - newarray(*Tipo*),anewarray(*Tipo*), crea un array tomando como primer argumento el tamaño y devuelve la referencia al mismo.
    - multianewarray(_, _), crea sucesivamente arrays de tamaños indicados en la lista que le pasan como argumentos de entrada.

- Clase:
    - instanceof(*Tipo*), devuelve 1 si el argumento de entrada es de tipo *Tipo* y 0 en caso contrario.
    - checkcast(*Tipo*), devuelve 1 si se puede hacer un casting con la variable de entrada.

- Campos:

- field(getfield, fieldref(*ObjectName, AttributeName, Tipo*, _)), obtiene el atributo *AttributeName* del objeto ObjectName y lo guarda en la salida.
- field(getstatic, fieldref(*ObjectName, AttributeName, Tipo*, _)), obtiene el atributo estático*AttributeName* del objeto ObjectName y lo guarda en la salida.
- field(putfield, fieldref(*ObjectName, AttributeName, Tipo,* _)), obtiene desde la variable de entrada un atributo y lo guarda en el atributo *AttributeName*del objeto ObjectName.
- field(putstatic, fieldref(*ObjectName, AttributeName, Tipo*, _)) , obtiene desde la variable de entrada un atributo y lo guarda en el atributo estático*AttributeName*del objeto ObjectName.
- init_vars, inicializa las variables de un bloque

- Excepciones:
  - athrow, copia la variable de entrada en una de salida (que será de excepción), sólo se usa este bytecode cuando la excepción se lanza explícitamente, antes hay que construir el objeto (algo que siempre ocurre antes de lanzar la excepción explícitamente).
  - ie_throw(*Excepcion*), crea un objeto del tipo *Excepcion* y devuelve la referencia como parámetro de salida, este bytecode es el resultado de la gestión implícita de excepciones y no se contabiliza durante el profiling, ya que las operaciones realizadas dependen de la máquina virtual utilizada.
  - assignment, copia variables de excepción a variables locales, eliminando la variable de entrada y viceversa. No se contabiliza durante el profiling, debido a que no son bytecodes de java, sino, que se han creado únicamente para utilizarse en la representación RBR.

Los Bytecodes deshechados, nop.

Como se mencionó antes, todo salto condicional o no condicional e invocaciones a métodos en la RBR se transforman en nop/2.

Algunas instrucciones se transforman en nop porque requieren conocer un resultado previo. Para esto, en la RBR se ha optado por dejar en el primer argumento el bytecode original y en el segundo una lista que indica si se debe o no considerar el coste de la operación nop.
Las instrucciones call y las guardas assr no se contabilizan, en ninguno de los análisis posibles pero, para llevar a cabo la suma de instrucciones, sí se contabilizan las instrucciones nop cuyo coste se indique que tiene que ser considerado.

Ejemplo de instrucción que hay que contabilizar puesto que presenta en el último parámetro la lista que ha de considerarse su coste, concretamente es un bytecode de comparación con cero:

nop(bytecode(88, cmpz(ge, 13), [s(1)], [], [stack_types=t([primitiveType(int)], []), nextaddr=89]), **[consider_cost(true)])**

Ejemplo conceptual de la traducción

Continuamos utilizando el mismo ejemplo que mostramos en la sección del Java Bytecode.

Ya vimos cómo se llevaba a cabo la ejecución de los bytecodes y su significado, ahora tomaremos tan sólo los bytecodes que java genera al llevar a cabo la compilación y veremos las reglas RBR equivalentes.

Las guardas que son ciertas se han omitido.

Utilizaremos la representación conceptual[12] de las reglas que permite ver que operaciones lleva a cabo y si se ejecuta después un bytecode, cuál es.

```
class A {
   int incr(int i) {
      return i+1;
   }
}

class B extends A {
   int incr(int i) {
      return i+2;
   }
}

class C extends B {
   int incr(int i) {
      return i+3;
   }
}

class Main {
   int add(int n,A o) {
      int res=0;
      int i=0;
      while (i<=n) {
         res=res+i;
         i=o.incr(i);
      }
   return res;
   }
}
```

12 ELVIRA ALBERT, PURI ARENAS and SAMIR GENAIM, Cost Analysis of Object-Oriented Bytecode Programs, página 17.

Clase A	Método incr.	
	A.incr (this, i, out) ←	A.incr$_1$ (this, i, out).
1: load 1	A.incr$_1$ (this, i, out) ←	s_1 := i,
2: push 1		s_2 := 1,
3: add		s_1 := s_1 + s_2,
4: return		out := s_1.

Clase B, extends A	Método incr.	
	B.incr (this, i, out) ←	B.incr$_1$ (this, i, out).
1: load 1	B.incr$_1$ (this, i, out) ←	s_1 := i,
2: push 2		s_2 := 2,
3: add		s_1 := s_1 + s_2,
4: return		out := s_1.

Clase C, extends B	Método incr.	
	C.incr (this, i, out) ←	C.incr$_1$ (this, i, out).
1: load 1	C.incr$_1$ (this, i, out) ←	s_1 := i,
2: push 3		s_2 := 3,
3: add		s_1 := s_1 + s_2,
4: return		out := s_1.

Clase Main	Método Add	
	Main.add (this, n, o, out) ←	Main.add$_1$ (this, n, o, res, i, out).
1: push 0	Main.add$_1$ (this, n, o, res, i, out) ←	s_1 := 0,
2: store 3		res := s_1,
3: push 0		s_1 := 0,
4: store 4		i := s_1,
		Main.add$_5$ (this, n, o, res, i, out).
5: load 4	Main.add$_5$ (this, n, o, res, i, out) ←	s_1 := i,
6: load 1		s_2 := n,
7: ifgt 17		nop(ifgt 17),
		Main.add$_5$(this, n, o, res, i, s_1, s_2, out).
Es una guarda	Main.add$_{c5}$(this, n, o, res, i, s_1, s_2, out) ←	s_1 > s_2,Main.add$_{17}$ (this, n, o, res, i, out).
Es una guarda	Main.add$_{c5}$(this, n, o, res, i, s_1, s_2, out) ←	s_1 ≤ s_2,Main.add$_8$ (this, n, o, res, i, out).
	Main.add$_{17}$ (this, n, o, res, i, out) ←	s_1 := res,
		out := s_1.
8: load 3	Main.add$_8$ (this, n, o, res, i, out) ←	s_1 := res,
9: load 4		s_2 := i,
10: add		s_1 := s_1 + s_2,
11: store 3		res := s_1,
12: load 2		s_1 := o,
13: load 4		s_2 := i,
14: invokevirtual A.incr:(I)I		nop(invokevirtual A.incr(I)I),

		Main.add$_{c8}$(this, n, o, res, i, s$_1$, s$_2$, out).
Es una guarda	Main.add$_{c8}$(this, n, o, res, i, s$_1$, s$_2$, out) ←	type(s$_1$,A),
		Main.add$_{14:A}$(this, n, o, res, i, s$_1$, s$_2$, out).
Es una guarda	Main.add$_{c8}$(this, n, o, res, i, s$_1$, s$_2$, out) ←	type(s$_1$,B),
		Main.add$_{14:B}$(this, n, o, res, i, s$_1$, s$_2$, out).
Es una guarda	Main.add$_{c8}$(this, n, o, res, i, s$_1$, s$_2$, out) ←	type(s$_1$,C),
		Main.add$_{14:C}$ (this, n, o, res, i, s$_1$, s$_2$, out).
Llama a incr de A	Main.add$_{14:A}$(this, n, o, res, i, s$_1$, s$_2$, out) ←	A.incr (s$_1$, s$_2$, s$_1$),
		Main.add$_{15}$ (this, n, o, res, i, s$_1$, out).
Llama a incr de B	Main.add$_{14:B}$(this, n, o, res, i, s$_1$, s$_2$, out) ←	B.incr (s$_1$, s$_2$, s$_1$),
		Main.add$_{15}$ (this, n, o, res, i, s$_1$, out).
Llama a incr de C	Main.add$_{14:C}$ (this, n, o, res, i, s$_1$, s$_2$, out) ←	C.incr (s$_1$, s$_2$, s$_1$),
		Main.add$_{15}$ (this, n, o, res, i, s$_1$, out).
15: store 4	Main.add$_{15}$ (this, n, o, res, i, s$_1$, out) ←	i := s$_1$,
16: goto 5		nop(goto 5),
17: load 3		Main.add$_5$ (this, n, o, res, i, out).
18: return		

Traducción completa que genera COSTA:

Se obtienen 18 bloques al solicitar al sistema COSTA que traduzca una llamada al método *add* de la clase Main. En esta sección se describirá lo que hace cada uno.
Se utilizará la misma numeración ordinal para referirse a los *simple bytecodes* de Java que se ha visto en la columna de la izquierda de la tabla de la sección anterior.

- Este primer bloque es el de llamada al método add de la clase Main. Se encarga de inicializar las variables del mismo y, como se puede ver, ejecuta los cuatro primeros bytecodes del método, apilando constantes y guardándolas en variables locales.

```
profiler/Main_add(ILprofiler/A;)I([l(0), l(1), l(2)],[s(1)],[e(1)],[A]) :=
    bytecode(-1, init_vars, [], [l(3), l(4)], [stack_types=t([], [])])
    bytecode(0, push(i, 0), [], [s(1)], [stack_types=t([], [primitiveType(int)]), nextaddr=1])
    bytecode(1, store(i, 3), [s(1)], [l(3)], [stack_types=t([primitiveType(int)], []), nextaddr=2])
    bytecode(2, push(i, 0), [], [s(1)], [stack_types=t([], [primitiveType(int)]), nextaddr=3])
    bytecode(3, store(i, 4), [s(1)], [l(4)], [stack_types=t([primitiveType(int)], []), nextaddr=5])
    call(block, 24, [[l(0), l(1), l(2), l(3), l(4)], [s(1)], [e(1)], [A]], [])
```

- El bloque 24 se ejecutará a continuación del identificado como *profiler/Main_add. E*n él hay dos llamadasa bloques. Primero se ejecutará el bloque 24_loop, cuyo nombre viene dado porque será una secuencia de bloques que ejecutará un bucle. Cuando acabe 24_loop, seguirá ejecutando el bloque 46.

```
24([l(0), l(1), l(2), l(3), l(4)],[s(1)],[e(1)],[A]) :=
    call(loop, 24_loop, [[l(1), l(2), l(3), l(4)], [l(3), l(4)], [l(3), l(4), e(2)], [B]], [])
    call(block, 46, [[l(0), l(1), l(2), l(3), l(4), e(2)], [s(1)], [e(1)], [B, A]], [])
```

– Este bloque contiene el cuerpo del bucle. En él se ejecutan los bytecodes 5 y 6 del método add. Éstos cargan las variables locales 4 y 1 en la pila. El bytecode 7, *goto,* ha sido transformado en *nop* -como ya se explicó antes- , por lo que será el bloque 47 el que implemente la guarda que lo controle.

```
24_loop([l(1), l(2), l(3), l(4)],[l(3), l(4)],[l(3), l(4), e(1)],[A]) :=
    bytecode(5, load(i, 4), [l(4)], [s(1)], [stack_types=t([], [primitiveType(int)]), nextaddr=7])
    bytecode(7, load(i, 1), [l(1)], [s(2)], [stack_types=t([primitiveType(int)], [primitiveType(int),
primitiveType(int)]), nextaddr=8])
    nop(bytecode(8, cmp(icmpgt, 27), [s(1), s(2)], [], [stack_types=t([primitiveType(int),
primitiveType(int)], []), nextaddr=11]), [consider_cost(true)])
    call(block, 47, [[l(1), l(2), l(3), l(4), s(1), s(2)], [l(3), l(4)], [l(3), l(4), e(1)], [A]], [])
```

– Si no se produjeron excepciones, la variable que controla el estado de la ejecución estará unificada con el valor *normal* y continuará ejecutando el bloque 25.

```
46([l(0), l(1), l(2), l(3), l(4), e(2)],[s(1)],[e(1)],[normal, A]) :=
    call(block, 25, [[l(0), l(1), l(2), l(3), l(4)], [s(1)], [e(1)], [A]], [])
```

– Si se produjeron excepciones la variable que controla el estado de la ejecución estará unificada con el valor *exception* y continuará ejecutando el bloque 28.

```
46([l(0), l(1), l(2), l(3), l(4), e(2)],[s(1)],[e(1)],[exception, A]) :=
    call(block, 28, [[l(0), l(1), l(2), l(3), l(4), s(1)], [], [e(1)], [A]], [])
```

– En este bloque la guarda actúa comparando dos variables enteras de pila. Si el resultado es que la primera es menor o igual que la segunda continúa ejecutando el bloque 26.

```
47([l(1), l(2), l(3), l(4), s(1), s(2)],[l(3), l(4)],[l(3), l(4), e(1)],[A]) :=
    assr(cmp(icmple), [s(1), s(2)], [])
    call(block, 26, [[l(1), l(2), l(3), l(4)], [l(3), l(4)], [l(3), l(4), e(1)], [A]], [])
```

– En este bloque la guarda actúa comparando dos variables enteras de pila. Si el resultado es que la primera es mayor que la segunda continúa ejecutando el bloque 24_loop_nexit.

```
47([l(1), l(2), l(3), l(4), s(1), s(2)],[l(3), l(4)],[l(3), l(4), e(1)],[A]) :=
    assr(cmp(icmpgt), [s(1), s(2)], [])
    call(block, 24_loop_nexit, [[l(1), l(2), l(3), l(4)], [l(3), l(4)], [], [A]], [])
```

– Si se llega al bloque 28, se ha producido una excepción y el objeto resultante de la misma se guarda en la variable de salida de excepción e(1).

```
28([l(0), l(1), l(2), l(3), l(4), s(1)],[],[e(1)],[exception]) :=
    bytecode(-2, assignment, [s(1)], [e(1)], [stack_types=t([refType(classType(A))],
[refType(classType(B))])])
```

– Este bloque está vacío porque es el caso en el cual la recursión acaba y no hay que ejecutar más bytecodes del bucle.

```
24_loop_nexit([l(1), l(2), l(3), l(4)],[l(3), l(4)],[],[normal]) :=
```

- El bloque 25 ejecuta los bytecodes 17 y 18 del método add, que cargan la tercera variable local en la pila y regresan del método respectivamente.
- El bytecode *return* no lleva a cabo ningún control del flujo. Sencillamente el bloque 25 no llama a ningún otro y por tanto acaba aquí su ejecución.

```
25([l(0), l(1), l(2), l(3), l(4)],[s(1)],[e(1)],[normal]) :=
    bytecode(27, load(i, 3), [l(3)], [s(1)], [stack_types=t([], [primitiveType(int)]), nextaddr=28])
    bytecode(28, return(i), [s(1)], [s(1)], [stack_types=t([primitiveType(int)], [primitiveType(int)]),
nextaddr=29])
```

- Ejecuta los bytecodes del 8 al 14 del método add. Éstos apilan la tercera y cuarta variable local, la suman, guardan el resultado en la tercera variable local y cargan después la segunda y cuarta variable local.
  El bytecode *invoke* se transforma en *nop*.
  Finalmente, continúa con el bloque 29.

```
26([l(1), l(2), l(3), l(4)],[l(3), l(4)],[l(3), l(4), e(1)],[A]) :=
    bytecode(11, load(i, 3), [l(3)], [s(1)], [stack_types=t([], [primitiveType(int)]), nextaddr=12])
    bytecode(12, load(i, 4), [l(4)], [s(2)], [stack_types=t([primitiveType(int)], [primitiveType(int),
primitiveType(int)]), nextaddr=14])
    bytecode(14, barithm(i, add), [s(1), s(2)], [s(1)], [stack_types=...]), nextaddr=15])
    bytecode(15, store(i, 3), [s(1)], [l(3)], [stack_types=t([primitiveType(int)], []), nextaddr=16])
    bytecode(16, load(a, 2), [l(2)], [s(1)], [stack_types=t([], [refType(classType(B))]), nextaddr=17])
    bytecode(17, load(i, 4), [l(4)], [s(2)], [stack_types=...]), nextaddr=19])
    nop(bytecode(19, invoke(virtual, methodref(profiler/A, incr(I)I, 1, 1, [primitiveType(int)],
primitiveType(int))), [s(1), s(2)], [s(1)], [stack_types=t([primitiveType(int), refType(classType(B))],
[primitiveType(int)]), nextaddr=22]), [consider_cost(false)])
    assr(nonnull, [s(1)], [])
    call(block, 29, [[l(1), l(2), l(3), l(4), s(1), s(2)], [l(3), l(4)], [l(3), l(4), e(1)], [A]], [])
```

- Llama al método virtual correspondiente. En este caso no se han creado guardas que comprueben el tipo del objeto y hagan llamadas al método *incr* de las clases B o C ya que COSTA, al generar los bloques, determinó que el tipo del objeto que se le pasó al invocar al método add era A y no B o C. Por eso tampoco creó reglas para los demás métodos.
  Cuando acabe la ejecución del método continurá ejecutando el bloque 45.

```
29([l(1), l(2), l(3), l(4), s(1), s(2)],[l(3), l(4)],[l(3), l(4), e(1)],[A]) :=
    nop(bytecode(19, invoke(virtual, methodref(profiler/A, incr(I)I, 1, 1, [primitiveType(int)],
primitiveType(int))), [s(1), s(2)], [s(1)], [stack_types=t([primitiveType(int), refType(classType(B))],
[primitiveType(int)]), nextaddr=22]), [consider_cost(true)])
    call(method(virtual), profiler/A_incr(I)I, [[s(1), s(2)], [s(1)], [e(2)], [C]], [])
    call(block, 45, [[l(1), l(2), l(3), l(4), s(1)], [l(3), l(4)], [l(3), l(4), e(1)], [C, A]], [])r=29])
```

- Continúa llamando al bloque 30 si el estado de la ejecución es *normal*.

```
45([l(1), l(2), l(3), l(4), s(1)],[l(3), l(4)],[l(3), l(4), e(1)],[normal, A]) :=
    call(block, 30, [[l(1), l(2), l(3), l(4), s(1)], [l(3), l(4)], [l(3), l(4), e(1)], [A]], [])
```

– La guarda actúa comprobando si la variable de excepción es un objeto *Throwable.* Sólo se llega a este bloque si la primera variable de la lista de excepciones que controla el estado de la ejecución está instanciada con el valor *exception.* Continúa ejecutando el bloque 24.

```
45([l(1), l(2), l(3), l(4), s(1)],[l(3), l(4)],[l(3), l(4), e(1)],[exception, A]) :=
    assr(instanceof([java/lang/Throwable]), [e(2)], [])
    bytecode(-3, assignment, [e(2)], [s(1)], [])
    call(block, 24_loop_eexit, [[l(1), l(2), l(3), l(4), s(1)], [], [l(3), l(4), e(1)], [A]], [])
```

– Este primer bloque es el de llamada al método incr de la clase A. Inicializa las variables del mismo y, como podemos ver, ejecuta todos los bytecodes del método add, que cargan la primera variable local, apilan la constante 1 y suman el resultado.
De nuevo encontramos la traducción de un bytecode *return.*
Al acabar llama al bloque 32.

```
profiler/A_incr(I)I([l(0), l(1)],[s(1)],[e(1)],[A]) :=
    bytecode(-1, init_vars, [], [], [stack_types=t([], [])])
    bytecode(0, load(i, 1), [l(1)], [s(1)], [stack_types=t([], [primitiveType(int)]), nextaddr=1])
    bytecode(1, push(i, 1), [], [s(2)], [stack_types=...]), nextaddr=2])
    bytecode(2, barithm(i, add), [s(1), s(2)], [s(1)], [stack_types=...]), nextaddr=3])
    bytecode(3, return(i), [s(1)], [s(1)], [stack_types=...]), nextaddr=4])
    call(block, 32, [[l(0), l(1), s(1)], [s(1)], [], [A]], [])
```

– Aquí podemos apreciar que los bloques no siguen un orden lineal. Algo lógico porque para computar es necesario que pueda romperse la linealidad en la ejecución del código. Ejecuta el bytecode 15 del método add que guarda el contenido de la pila en la cuarta variable local y convierte *goto* en *nop.*
Al continuar la ejecución del bloque *24_loop* el efecto es el mismo que el de ejecutar el salto definido por *goto.* Éste es el ciclo que permite implementar el bucle.

```
30([l(1), l(2), l(3), l(4), s(1)],[l(3), l(4)],[l(3), l(4), e(1)],[A]) :=
    bytecode(22, store(i, 4), [s(1)], [l(4)], [stack_types=t([primitiveType(int)], []), nextaddr=24])
    nop(bytecode(24, goto(5), [], [], [stack_types=t([], []), nextaddr=27]), [consider_cost(true)])
    call(block, 24_loop, [[l(1), l(2), l(3), l(4)], [l(3), l(4)], [l(3), l(4), e(1)], [A]], [])
```

– A continuación hay dos reglas sin bytecodes que determinan el fin de la ejecución, la primera del bucle poque se produjo una excepción y la segunda del método incr de la clase A.

```
24_loop_eexit([l(1), l(2), l(3), l(4), s(1)],[],[l(3), l(4), e(1)],[exception]) :=
```

```
32([l(0), l(1), s(1)],[s(1)],[],[normal]) :=
```

EL SISTEMA COSTA

Nuestro profiler se encuentra integrado dentro del sistema COSTA (COSt and Termination Analyzer for Java Bytecode) desarrollado en colaboración entre las universidades Complutense y Politécnica de Madrid.

Descripción

El estudio de la complejidad computacional es un area fundamental en la Ingeniería informática, tratando de determinar la cantidad de recursos necesarios para ejecutar un determinado algoritmo en función de los valores de entrada para el mismo.

El ámbito de actuación del sistema COSTA está centrado en dicho estudio.

Su objetivo es realizar un análisis de la terminación y el coste de programas bytecode de Java (puede analizar tanto código para Java SE como para Java ME)[13]. Sin embargo no realiza el análisis directamente sobre el código bytecode, sino que transforma este último en una representación de reglas intermedia llamada RBR (Rule Based Representation) explicada con detalle en el punto anterior, más cómoda para su utilización en Prolog, lenguaje utilizado por el sistema para desarrollar su funcionalidad.

El análisis del coste trata de estimar la cantidad de recursos consumidos por un programa en tiempo de ejecución, mientras que el objetivo del análisis de la terminación de un programa trata de probar si un programa termina para cualquier entrada.

Tener tanto el análisis de coste como el de terminación en una misma herramienta es interesante, ya que dichos análisis comparten muchos aspectos y así una gran parte del analizador es común a ambos.

COSTA es un analizador estático que toma dos entradas, el programa en formato bytecode de Java que se quiere analizar y un modelo de coste para aplicar a dicho programa. El modelo de coste definirá el tipo de medida de recursos que se desea hacer sobre el programa proporcionado.

El sistema tiene definidos varios modelos de coste, como por ejemplo el consumo de heap, el número de instrucciones bytecode ejecutadas y el número de llamadas a un método específico que puede ser definido por el usuario.

Una vez seleccionados el programa de entrada y el modelo de coste deseado, el sistema consigue obtener una expresión de coste de ejecución del programa con respecto al modelo de coste seleccionado, en función de los argumentos de entrada. El sistema también consigue demostrar la terminación del programa.

COSTA se basa en la aproximación clásica al análisis estático de costes, que consta de dos fases. Primero, dado un programa y una descripción del recurso que se desea medir, el análisis genera relaciones de coste, que son conjuntos de ecuaciones recursivas. Despues, se encuentran soluciones para dichas ecuaciones, si es posible.

13 ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, GERMÁN PUEBLA, DAMIANO ZANARDINI, Resource Usage Analysis and Its Application to Resource Certification, página 284.

Arquitectura[14]

COSTA está implementado en Prolog y puede manejar un gran subconjunto del bytecode de Java (excepto concurrencia e inicialización de atributos estáticos maneja todo). Dicho subconjunto difiere del lenguaje bytecode simplificado que se usa en la formalización en varios aspectos:

- El conjunto de instrucciones incluye distintas variantes de la misma instrucción, dependiendo del tipo de operandos (iload, aload, etc. son variantes de la funcionalidad load con argumentos de distinto tipo).
- Las llamadas a procedimientos pueden tomar la forma de invokevirtual para invocación dinámica, invokespecial para invocación especial e invokestatic para los métodos estáticos.
- Los metodos no son forzados a tener un valor de retorno.
- Las excepciones estan soportadas, ya sean lanzadas explicitamente en el código o resultado de violaciones semánticas. Maneja excepciones internas (aquellas asociadas a los bytecodes según la especificacion de la máquina virtual de Java), excepciones lanzadas (bytecode athrow) y excepciones propagadas.

La siguiente figura muestra la arquitectura del sistema.

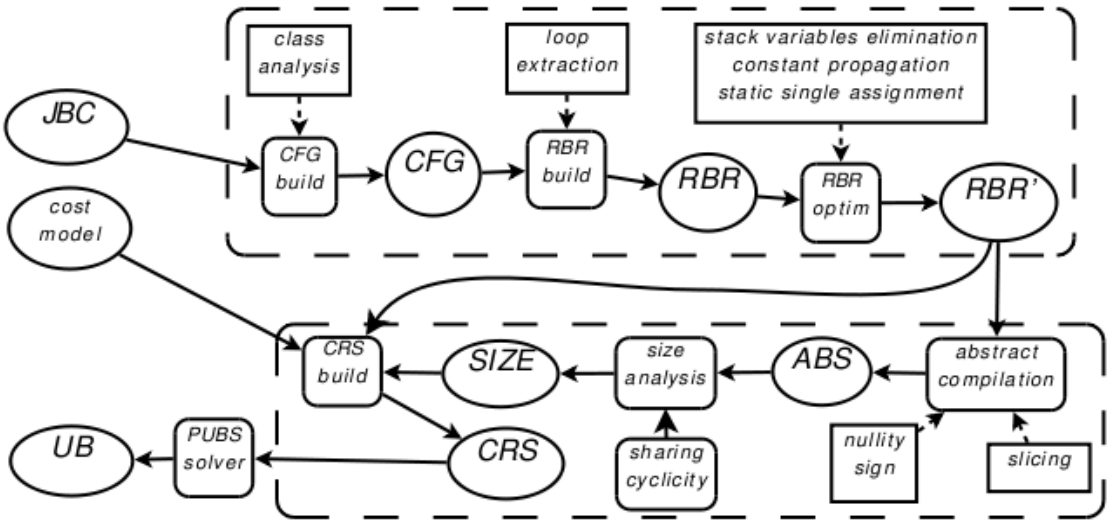

Imagen 1

Los recuadros discontinuos representan las dos partes principales del análisis: la transformación del bytecode de entrada (JBC) en la representación basada en reglas (RBR') y la realización del análisis basado en dicha representación obteniendo el límite superior del coste de ejecución (UB).

Las entradas y salidas del sistema están representadas a la izquierda: la parte de transformación de bytecode en RBR toma como entrada el programa bytecode (JBC), mientras que el módulo

14 ELVIRA ALBERT, PURI ARENAS and SAMIR GENAIM, Cost Analysis of Object-Oriented Bytecode Programs, página 42.

que se encarga de realizar el análisis toma como entrada el modelo de coste (cost model), así como también la salida del modulo de transformación (RBR'), es decir la representación del programa bytecode en formato RBR.

La salida que produce el sistema es un límite superior del coste (UB).

Los recuadros redondeados se corresponden con los pasos principales del proceso (por ejemplo CFG build), mientras que las elipses representan el resultado de dichos pasos.

En una primera fase el programa de entrada (JBC) se transforma en la representación basada en reglas (RBR) a través de la construcción de un grafo de control de flujo (CFG), grafo que representa el flujo de ejecución que va a seguir el programa. Varias técnicas como el análisis de clases (class analysis), la extracción de bucles (loop extraction), la eliminación de variables de pila (stack variables elimination), la asignación única estatica (static single assignment) y la propagación de constantes (constant propagation) hacen este proceso más eficiente y preciso .

El analisis de clases trata de computar el conjunto de instancias método que pueden ser invocadas por una llamada virtual. Esta información es particularmente útil a la hora de construir el grafo de control del flujo, ya que permite excluir varias instancias método.

La asignación única estática consiste en renombrar las variables con el fin de garantizar que cada variable solo es asignada una vez. Esto ayuda a propagar las constantes a traves de las reglas de la RBR via unificación. Conocer que una variable es realmente una constante en un determinado punto del programa puede ser muy util a la hora de realizar ciertas operaciones. Por ejemplo puede mejorar el tamaño de abstracción de idiv (cuando el divisor es constante), lo cual es crucial para inferir un limite superior de coste preciso para ejemplos como la búsqueda binaria.

Otra técnica importante es la extracción de bucles del grafo de control de flujo. Cuando se analiza bytecode, reconocer estructuras iterativas en el grafo de control de flujo puede evitar la pérdida de información que se deriva de un flujo de control no estructurado. En particular, detectar bucles anidados permite hacer el razonamiento bucle a bucle, lo que hace que hallar un limite de coste para la ecuación correspondiente sea más fácil.

La extracción de bucles se aplica al grafo de control de flujo inicial con el fin de separar subgrafos correspondientes a los bucles. COSTA implementa un algoritmo que ademas de extraer bucles con una única entrada, también tienen una única salida (para evitar multiples ramas de retorno de los bucles). Cuando un bucle es extraido, el correspondiente subgrafo es reemplazado por una nueva instrucción de llamada a bucle. Además un nuevo grafo de control de flujo es generado para cada subgrafo. Así pues después de este paso, hay un grafo de control de flujo para la entrada del metodo y otro para cada bucle. Debido a este diseño de la RBR, las llamadas a bucles se tratan de la misma forma que las llamadas a métodos.

En la segunda fase se realiza el análisis de coste de la RBR que se ha obtenido en la primera fase. La compilación abstracta (Abstract compilation) prepara la RBR que se recibe como entrada para el analisis de tamaño (Size analysis).

COSTA utiliza una serie de técnicas de análisis estático (sign, nullity) que ayudan a excluir ciertos comportamientos del programa, lo cual ayuda a agrupar reglas que resultarían en una serie de reglas sin ninguna ramificación en una sola regla.

Para detectar estructuras con ciclos COSTA utiliza varios análisis (sharing, acyclity), ya que los programas con ciclos pueden no tener un limite superior de ejecución, es decir puede ser infinito.

Tras esto, COSTA crea unas relaciones de coste (CRS) teniendo en cuenta el modelo de coste seleccionado, que tras ser analizadas por una herramienta de resolución de limites superiores (PUBS solver), devuelven como salida el limite superior de ejecución (UB) del programa de

entrada con respecto al modelo de coste de entrada.

Aplicación

Los resultados de los análisis llevados a cabo por COSTA son de gran utilidad para los desarrolladores. Gracias a ellos se puede garantizar que un programa no va a consumir demasiado tiempo o recursos en su ejecución y que va a terminar.

Otro aspecto en el que su aplicación es de gran utilidad es en la optimización de programas. Los límites superiores de coste de ejecución obtenidos para el programa según los distintos modelos de coste pueden ser utilizados para decidir entre diversas implementaciones alternativas para un programa de forma que se optimice el consumo de recursos.

Pensando precisamente en prestar apoyo a los desarrolladores, el equipo de COSTA ha desarrollado un plugin para la conocida herramienta de programación ECLIPSE que permite utilizar el analizador de manera cómoda y sencilla mientras se esta desarrollando software[15].

Con respecto al ámbito de lenguajes de programación en los que puede ser utilizado el sistema COSTA, es interesante remarcar que no sólo es aplicable a programas escritos en código bytecode Java. Aunque su principal utilización está centrada en programas escritos en dicho lenguaje de programación, las técnicas utilizadas en el sistema no son sólo aplicables a dicho lenguaje, sino que pueden ser directamente aplicadas para inferir relaciones de coste en otros lenguajes de programación imperativa orientados a objetos, no necesariamente en formato bytecode.

Relación entre COSTA y el profiler

Como hemos visto en las secciones anteriores las principales funcionalidades del sistema COSTA son las siguientes:

- Obtener el coste de ejecución de un programa respecto a un determinado modelo de coste.
- Comprobar la terminación de un programa.

La función de nuestro profiler está encuadrada en la primera, ya que como hemos visto anteriormente proporciona el coste de ejecución de programas en formato bytecode conforme a tres modelos de coste: número de instrucciones bytecode ejecutadas, número de llamadas a un determinado método definido por el usuario y número de objetos creados en el heap.

Para realizar esto nuestro profiler se ha tenido que servir de código propio del sistema COSTA, principalmente del módulo que ejecuta la primera fase vista en el apartado “Arquitectura” y que transforma el codigo bytecode de un programa en una representación basada en reglas (RBR) optimizada. Dicha RBR optimizada será la entrada que reciba nuestro profiler y, a partir de la cual teniendo en cuenta el modelo de coste que desee aplicar el usuario, producirá como salida el coste que supone el programa para cada uno de los parámetros seleccionados por el mismo.

15 ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, GERMÁN PUEBLA, DAMIANO ZANARDINI, Resource Usage Analysis and Its Application to Resource Certification, página 283.

INTERFAZ POR CONSOLA

Aunque la interfaz que por defecto se utilizará para hacer uso del sistema será la interfaz web, es posible que haya casos en los que sea más conveniente utilizar la interfaz por consola. En esta sección se detalla el uso de ésta.

La interfaz propiamente dicha se encuentra en el script con nombre 'profiler' que se encuentra en la carpeta raiz del sistema. Es por ello necesario que dicho script se encuentre con permisos de ejecución en el momento que el usuario quiera hacer uso de él.

La sintaxis es:

./profiler -c 'clase' -e 'metodo-de-entrada' ['opciones'] [-a 'argumentos-entrada']

'clase' debe seguir la sintaxis 'paquete'/'nombre-clase', donde 'nombre-clase' debe corresponderse con un archivo de codigo de bytes con extensión .class contenido en una de las carpetas a las que el CLASSPATH del sistema apunta y pertenecer al paquete 'paquete'.

'metodo-entrada' debe seguir la sintaxis de Java para referenciar a métodos y dicha referencia debe apuntar a un método contenido en el código de bytes especificado en el parámetro 'clase' anterior.

A continuación se enumeran las opciones que se le pueden pasar al script y se detalla su funcion:

- *i*: conteo de instrucciones; indica al intérprete que debe contabilizar el número de instrucciones ejecutadas y presentar la información al finalizar la ejecución.

-*m ['metodo-a-contabilizar']*: conteo de métodos; indica al intérprete que debe contabilizar el número de llamadas a todos los métodos utilizados en la ejecución y presentar la información al finalizar ésta. En caso de proporcionar el nombre de un método en 'metodo-a-contabilizar' el intérprete se limitará a contabilizar las llamadas a ese método en concreto.

-*o*: conteo de objetos; indica al intérprete que debe contabilizar el número de objetos creados durante la ejecución y presentar la información al finalizar ésta.

-*r* 'tipo-retorno': indica en 'tipo-retorno' el tipo, en caso de que lo haya, del valor de retorno.
'tipo-retorno' puede ser:

1. void: sin valor de retorno.
2. simple: tipos simples integer, float, double y long.
3. char: tipo char.
4. string: tipo String.

El sistema, por defecto, establece que no hay valor de retorno.

-t: modo traza; indica al intérprete que se quiere depurar la ejecución y para ello el sistema presentará la interfaz de depuración al comenzar ésta primera.

En caso de que exista paso de argumentos de entrada deberá especificarse siempre tras las opciones haciendo uso el párametro *-a* y tantos pares 'tipo-argumento' 'valor-argumento' como argumentos sean necesarios.

Los tipos posibles se detallan a continuación y se corresponden con los tipos Java mencinados en la sección *Detalles de la implementación*.

- 'I': integer
- 'Z': boolean
- 'F': float
- 'C': char
- 'D': double
- 'J': long
- 'Ljava/lang/String;': String
  El contenido del String debe ir entre comillas simples.
- '[Ljava/lang/String;': Array de String
  El formato para el paso de cada String es: 'string1|string2|..|stringn'

Ejemplos de uso:

```
./profiler -c 'profiler/profiler1' -e 'main()V'
./profiler -c 'profiler/profiler3' -e 'fact(I)I' -r simple -a I 4
./profiler -c 'profiler/inargs2' -e 'compareBoolean(Z)Z' -r simple -a Z true
./profiler -c 'profiler/inargs2' -e 'compareFloat(F)Z' -r simple -a F 1.1
./profiler -c 'profiler/inargs2' -e 'compareChar(C)Z' -r simple -a C 'a'
./profiler -c 'profiler/inargs2' -e 'compareLong(J)Z' -r simple -a J 100
./profiler -c 'profiler/inargs2' -e 'compareDouble(D)Z' -r simple -a D 100.1
./profiler -c 'profiler/string3' -e 'main(Ljava/lang/String;)V' -a 'Ljava/lang/String;' 'string 1'
```

INTERFAZ GRÁFICA

Como hemos explicado anteriormente, nuestro profiler está programado íntegramente en Prolog. Para que no fuese necesario para el usuario tener un conocimiento básico de Prolog y hacer más sencillo y cómodo su uso, se decidió desarrollar una interfaz gráfica web en lenguaje PHP que conectase directamente con el profiler, de forma que el usuario pudiera utilizarlo sin necesidad de conocimientos técnicos de manera cómoda y rápida.

Estructura

Para la realización de la interfaz gráfica se tomó como referencia la interfaz web de COSTA, que está disponible en la dirección http://costa.ls.fi.upm.es/~costa/costa/costa.php.

La estructura de la interfaz gráfica sigue un esquema simple formado por cuatro pantallas:

- Pantalla inicial.
- Pantalla de selección de archivo.
- Pantalla de selección del metodo de entrada y modelo de coste.
- Pantalla de ejecución y presentación de resultados.

La apariencia de todas las ventanas ha sido configurada mediante la hoja de estilos cost.css que se encuentra en el directorio de la interfaz.

Como ayuda a la explicación de la interfaz gráfica se tomará como punto de partida el siguiente programa Java, que será el que se ejecute:

```
package profiler;

public class prueba_Interfaz{

        public static int main(int opc, String[] arg){
                int i=prueba(opc,arg[0],arg[1]);
                return i;
        }

        public static int prueba(int i,String a, String b){
                int res;
                if (i==0){
                        res=factorial(5);
                }
                else{
                        boolean bo= iguales(a,b);
                        if (bo==false){
                                res=0;
                        }
                        else{
                                res=1;
                        }
                }
                return res;
```

```
	}

	public static int factorial(int i){
		if (i<=1)
			return 1;
		else
			return i * factorial(i-1);
	}

	public static boolean iguales(String a, String b){
		return a.equals(b);
	}

}
```

A continuación se describen las distintas pantallas con sus funcionalidades.

Pantalla inicial

Es la pantalla que aparece al iniciar la interfaz. Contiene una breve descripción de la función del profiler y un enlace a las pantallas inicial y de selección de archivo. Estos enlaces aparecerán en todas las pantallas de la interfaz gráfica, dando al usuario la posibilidad de volver a empezar el proceso de ejecución del profiler en cualquier momento.

La siguiente imagen muestra la pantalla inicial.

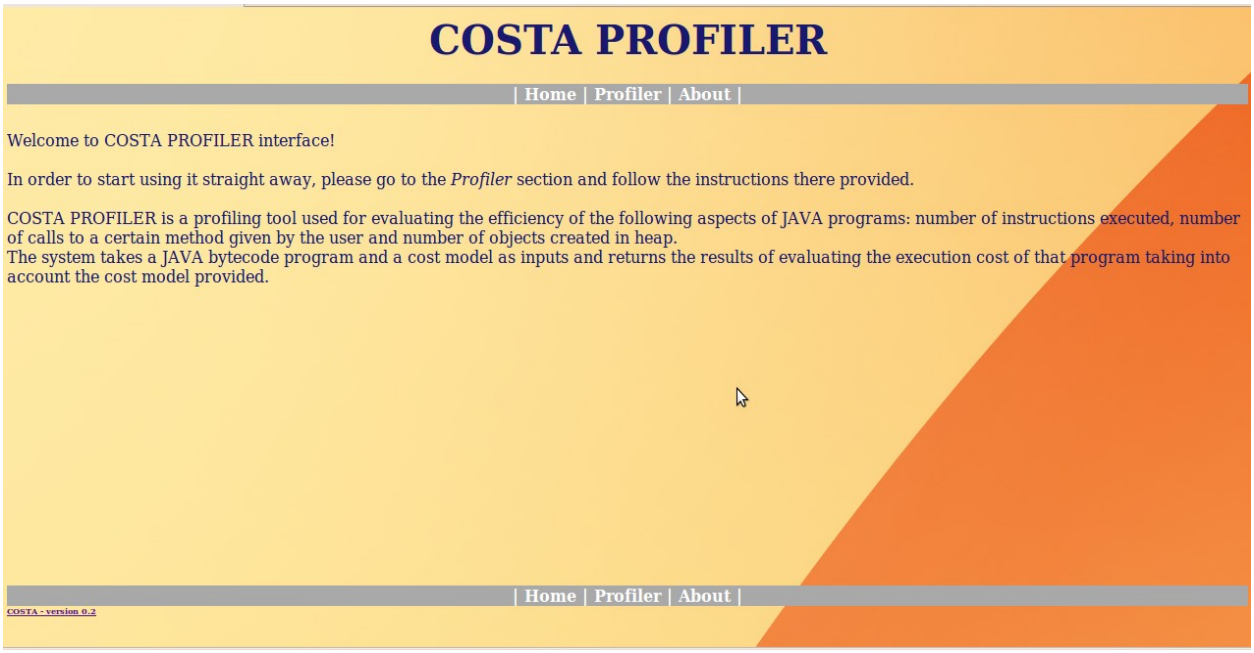

Imagen 2

Como se aprecia en la imagen, en la parte superior, debajo del título aparecen tres enlaces: “Home”, “Profiler” y “About” que conducen a la pantalla inicial, a la pantalla de selección de archivo y a una pantalla de información sobre el profiler respectivamente.

Para iniciar el proceso de ejecución del profiler el usuario deberá hacer click en el link “Profiler”, lo que le llevará a la siguiente pantalla.

La pantalla inicial se corresponde con el archivo costa.php del directorio de la interfaz.

Pantalla de selección de archivo

En esta pantalla el usuario debe seleccionar la primera de las entradas necesarias para el profiler: el archivo que contiene el programa Java sobre el que se desea realizar el análisis. Dicho archivo debe ser un archivo de extensión *.class*.

La pantalla está formada por un formulario que contiene un botón "Examinar" que, una vez pulsado, lanza un explorador de archivos para que el usuario elija el archivo que desee y un boton "Upload" para enviar el resultado del formulario (el archivo sobre el que hacer el perfilado) a la siguiente pantalla (Selección del método de entrada y del modelo de coste).

La siguiente imagen muestra la pantalla de selección de archivo.

Imagen 3

Una vez seleccionado el archivo que contiene el código sobre el que se desea ejecutar el profiler el usuario debe pulsar el botón "Upload" tras lo que será dirigido a la siguiente pantalla.

Si el usuario hace click sobre el botón "Upload" antes de elegir un archivo, o tras haberlo elegido, pero teniendo éste una extensión no válida para el profiler, se mostrará un mensaje de error y no se pasará a la pantalla de selección de método de entrada y modelo de coste.

Esta pantalla se corresponde con el archivo analizer.php del directorio de la interfaz.

Pantalla de selección de método de entrada y de modelo de coste

En esta pantalla el usuario debe seleccionar el método inicial del programa elegido para hacer el profiling, así como la segunda de las entradas del profiler, esto es, el modelo de coste que desea aplicar.

Lo primero que aparece en esta pantalla es un campo mostrando el archivo que ha sido seleccionado en la pantalla anterior.

Tras esto, aparece una lista con los métodos que contiene el archivo que ha sido elegido para que el usuario seleccione el que desee como método de entrada (método que se ejecutará al iniciar el

profiler).

Sólo se podrá seleccionar un único método.

Si reciben argumentos, los métodos aparecerán seguidos de campos para rellenar con los argumentos con los que se desea ejecutar el método, precedidos del tipo de dichos argumentos. Para los arrays la forma de rellenar el campo será separando los elementos con el carácter '|'. Por ejemplo para un array de los String "prueba1", "prueba2" y "prueba3" la manera correcta de rellenar el campo del array será: prueba1|prueba2|prueba3

Las tipos de datos que aceptan los métodos como argumentos y sus abreviaturas son las siguientes:

- 'I': integer
- 'Z': boolean
- 'F': float
- 'C': char
- 'D': double
- 'J': long
- 'Ljava/lang/String;': String
- '[Ljava/lang/String;': Array de String

Para extraer los métodos del archivo introducido por el usuario se utilizan las funciones *methods()* para extraer los métodos del archivo y *show_methods()* para mostrarlos en pantalla, definidos en el archivo config.php del directorio de la interfaz. La función *methods()* a su vez utiliza el script *get_methods* del directorio, que fue desarrollado para la interfaz de COSTA y que reutilizamos para extraer la lista de métodos de un archivo.

La signatura de los métodos extraidos del archivo introducido por el usuario mediante el script *get_methods* es la siguiente:

nombre-metodo(tipo-argumento-1,...,tipo-argumento-n)tipo-resultado.

En la interfaz de COSTA no estaba incluida la funcionalidad de poder ejecutar métodos pasándoles el valor de sus argumentos. Gracias a la signatura de los métodos devueltos por el script *get_methods*, ha resultado sencillo parsear, mediante funciones php, el String que contenía el método para poder introducir el tipo y número de campos correspondientes a los parámetros de cada método. Esto se realiza mediante la función *parseArgs()* del archivo config.php del directorio de la interfaz que, para cada método, genera la correspondiente secuencia de campos con sus tipos en función de los parámetros del mismo.

Tras esto aparecerán los tres modelos de coste disponibles:

- Número de instrucciones bytecode ejecutadas
- Número de objetos creados en el heap
- Número de llamadas a un método

El usuario debe seleccionar los que desee aplicar. En el caso de número de llamadas a un método aparece un combobox con los métodos de la clase para que el usuario escoja el método que desee contar. Si selecciona el componente vacío del combo, se contarán todas las llamadas a

cualquier método.

El usuario puede elegir varios modelos de coste para aplicar a la vez.

Esta pantalla se corresponde con el archivo costmodel.php del directorio de la interfaz.

Las siguientes imagenes muestran la pantalla de selección de método de entrada y modelo de coste. En la primera el usuario ha elegido contar solo el método *"factorial"* mientras que en la segunda ha elegido contar todos los métodos seleccionando el campo vacío del combo.

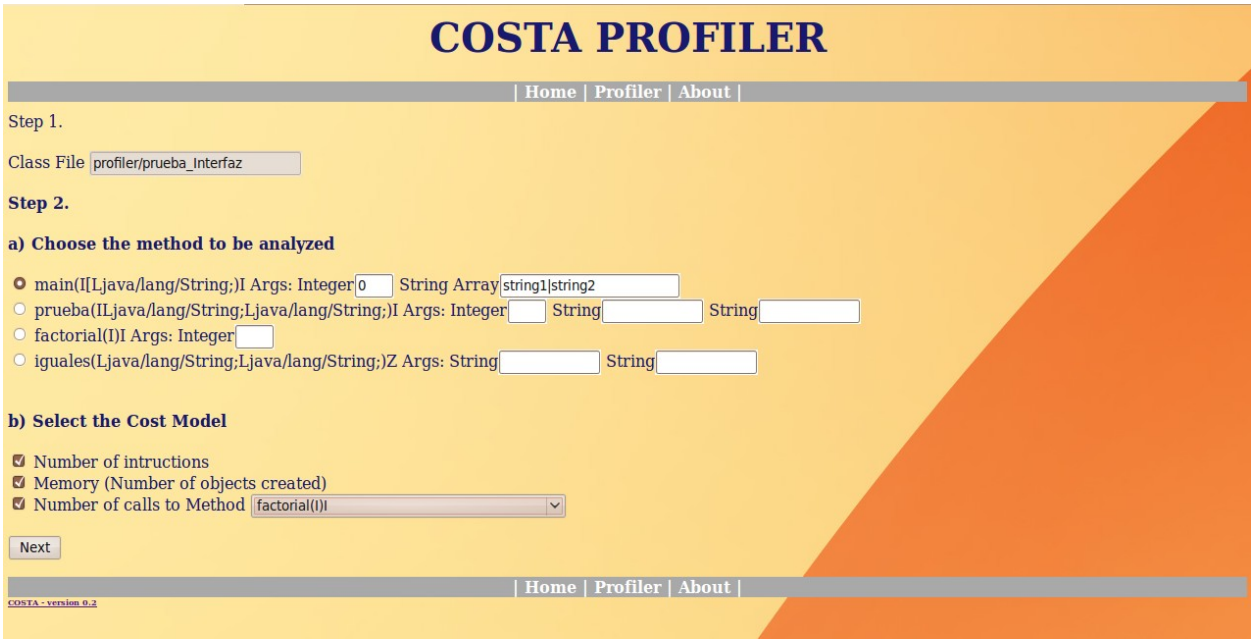

Imagen 4

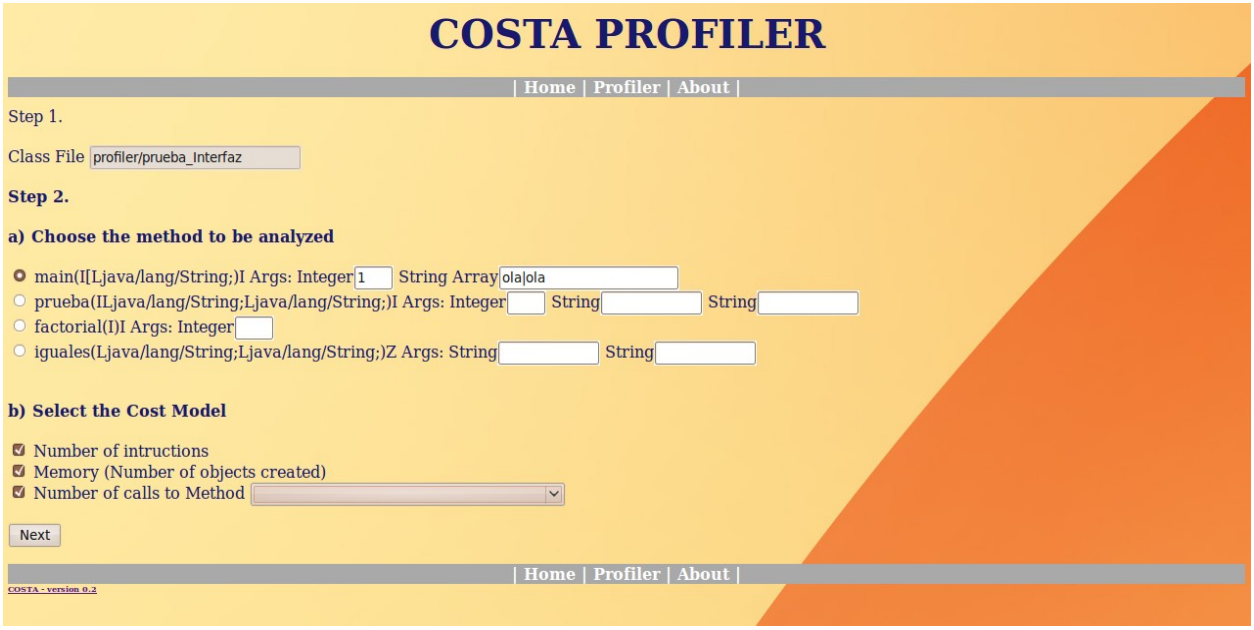

Imagen 5

Tras haber seleccionado un método de entrada y los modelos de coste deseados, el usuario debe pulsar el botón “Next” tras lo que será dirigido a la pantalla de ejecución y presentación de resultados.

Pantalla de ejecución y presentación de resultados

Esta pantalla es la encargada de enlazar la interfaz gráfica con el profiler. Para ello nos servimos del script “profiler” que creamos con ese objetivo. Mediante la función shell_exec() del lenguaje php ejecutamos el script con los parametros correctos en función de las elecciones del usuario en las pantallas anteriores y presentamos los resultados. Dichos parámetros se obtienen mediante la función build_args() del archivo config.php del directorio de la interfaz que, en función del método que haya sido escogido por el usuario para la ejecución, construirá la llamada al script del profiler con los argumentos adecuados.

Así pues en esta pantalla son presentados al usuario los resultados del profiler aplicados al archivo, método de entrada y modelos de coste que ha elegido.

Esta pantalla se corresponde con el archivo analyse.php del directorio de la interfaz.

Las siguientes imagenes nos muestran las pantallas de resultados correspondientes a ejecutar desde la situación definida en las imagenes 4 y 5 respectivamente.

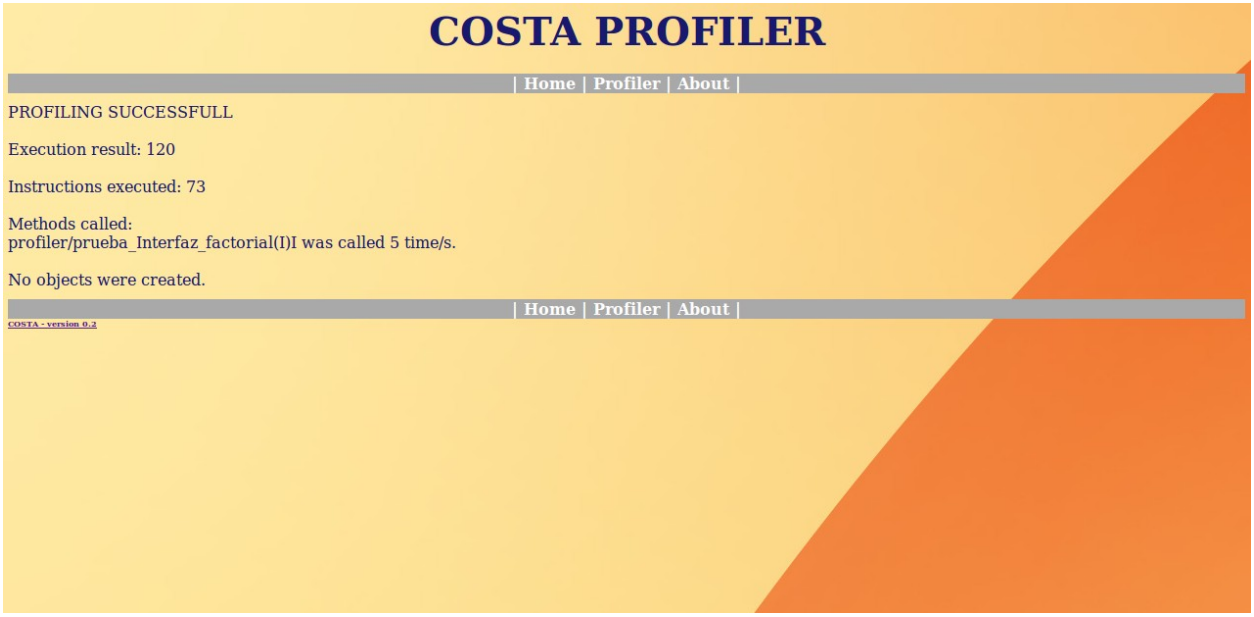

Imagen 6

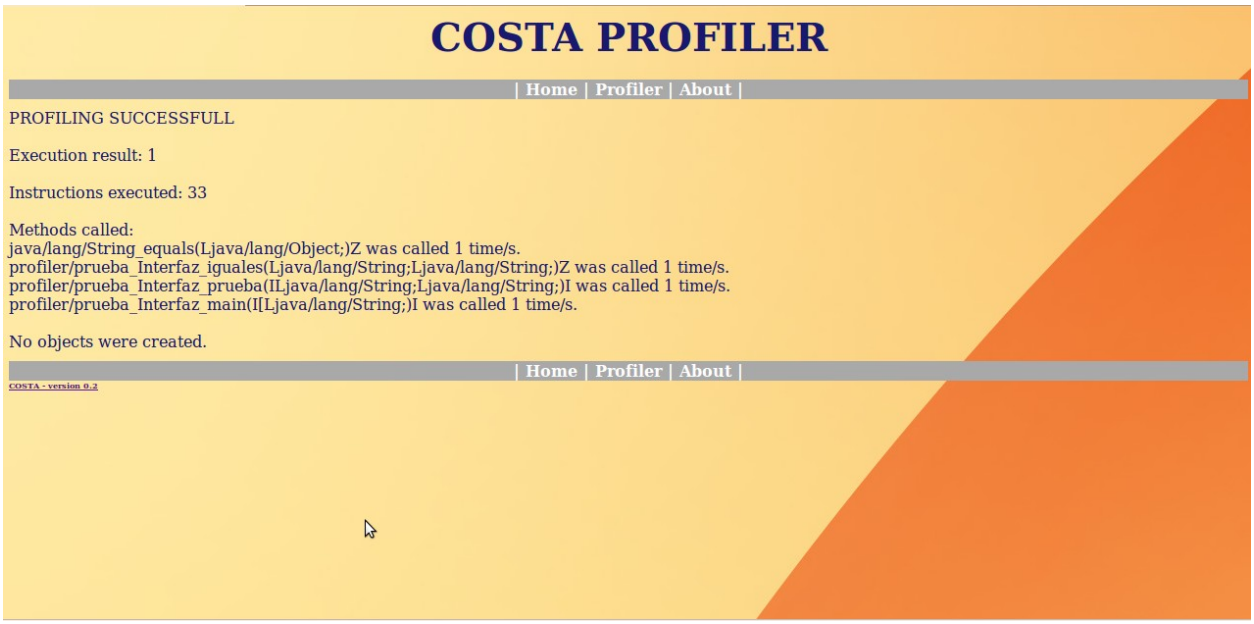

Imagen 7

Como podemos observar en la imagen 6, tras seleccionar que solo se tenga en cuenta el método "factorial" a la hora de hacer el conteo de métodos, los resultados sólo muestran las veces que se ha llamado a dicho método. En la imagen 7 se puede ver que, al haber seleccionado contar todos los métodos, aparecen todos los métodos que han sido llamados, no solo de la clase "prueba_Interfaz", sino que también se ha ejecutado el método *String_equals* de la clase Object.

Información importante para el correcto funcionamiento de la interfaz

Debido a que la interfaz se ejecuta desde un servidor php, el usuario que la ejecuta es el usuario de php, esto es "www-data". Por lo tanto, como son necesarios varios accesos a directorios y archivos de la estructura de directorios del sistema COSTA, cuyo propietario suele ser otro distinto a "www-data" es necesario dar permisos de lectura, escritura y ejecución al usuario "www-data" en toda la estructura de directorios de COSTA para evitar problemas de permisos.

Asimismo es necesario que, a la hora de instalar la interfaz del profiler en un PC, se incluyan en el archivo del que leen al iniciarse las shells que no pertenecen al usuario que tenga instalado COSTA, normalmente /etc/bashrc, la definición de las variables $PATH y $CLASSPATH que se encuentra en el archivo /USER/.bashrc (siendo USER el nombre del directorio del usuario que tenga instalado COSTA), ya que la orden shell_exec() del lenguaje php ejecuta una orden en una shell cuyo propietario es el usuario "www-data" y necesita que estén bien definidas dichas variables.

DETALLES DE LA IMPLEMENTACIÓN

Introducción

Todo análisis comienza con la inicialización del intérprete. Este proceso consiste en establecer un estado inicial para la memoria, reiniciar todos los contadores utilizados para realizar el análisis de los recursos utilizados e instanciar en la memoria algunos objetos fundamentales para la correcta ejecución de todo código de bytes.

Seguidamente, si procede, se produce el mapeo de los argumentos de entrada en la memoria y se hace una llamada al sistema COSTA para que transforme el código de bytes contenido en la clase especificada al inicio en una representación recursiva basada en reglas. Lo siguiente es buscar la regla inicial en la rbr que contenga la primera instrucción a ejecutar en función del método de entrada especificado al inicio.

Tras esto, el intérprete sigue el flujo adecuado para el programa en cuestión, ejecutando instrucciones, teniendo en cuenta aserciones, realizando las llamadas a bloques pertinentes y, llegado el momento, ejecutando la última instrucción del código de bytes.

Finalmente, y teniendo en cuenta si la máquina se encuentra en un estado normal o de excepción, presenta un informe con la información sobre el análisis que se le requirió al comienzo.

Esquema del código fuente

- *batch_tests*: script de consola que contiene una llamada al intérprete por cada ejemplo de prueba utilizado durante la implementación. Básicamente se ha utilizado como batería de pruebas para evitar que cambios en el intérprete produjeran errores en la ejecución de programas ya comprobados.

- *costa_profiler.pl*: fichero principal de la implementación. Contiene el predicado principal del intérprete 'costa_profiler', el tratamiento de las llamadas a bloques, el tratamiento de las aserciones, el tratamineto de las instrucciones y el conteo de los recursos a analizar.

- *decompiler.pl*: fichero que contiene todas las funciones del sistema COSTA que el intérprete necesita para su correcto funcionamiento. Básicamente se encarga de hacer las llamadas necesarioas al sistema COSTA para transformar un códgo de bytes en una representación recursiva basada en reglas.

- *profiler*: script de consola que contiene la interfaz por consola. Su funcionamiento se ha explicado en la sección *Interfaz por consola*.

- *profiler_arrays.pl*: fichero que contiene las reglas relacionadas con el tratamiento de arrays tanto de tipos básicos como de objetos así como de una dimensión o de varias.

- *profiler_heap.pl*: fichero que contiene las reglas relacionadas con el tratamiento de la memoria del sistema, tanto de los marcos o frames pertenecientes a los bloques como del heap, así como los predicados que se han establecido para la correcta organización de ésta.

- *profiler_native_methods.pl*: fichero que contiene implementaciones para algunos de los métodos nativos dependientes de la plataforma y utilizados por Java y que son necesarios para el correcto funcionamiento del interprete.

- *profiler_static_fields.pl*: fichero que contiene las reglas relacioandas con el acceso y tratamiendo de los campos estáticos.

Bloques de instrucciones

Como se ha explicado en la sección *Representación recursiva basada en reglas (RBR), t*oda instrucción contenida en la representación recursiva pertenece a un bloque caracterizado por un nombre. Cada bloque puede contener bytecodes, aserciones o llamadas a otros bloques.

A su vez, cada bloque tiene asociado un marco o frame que contiene las variables locales o de pila relativas a ese bloque. Todo marco o frame asociado a un bloque es independiente de los demás.

El nombre del bloque inicial que el intérprete utiliza como punto de partida siempre tiene el formato 'clase'_'metodo-de-entrada' donde 'clase' y 'metodo-de-entrada' se corresponden con las opciones explicadas anteriormente y pasadas al intérprete durante la llamada inicial.

Una vez iniciada la ejecución, el intérprete continuará hasta que se ejecute la última instrucción de un bloque y ésta no sea una llamada a otro bloque o se llegue a un bloque que no contenga instrucciones. En este punto el intérprete producirá vuelta atrás buscando la siguiente instrucción, aserción o llamada no procesada o finalizando la ejecución en caso de no encontrar ninguna de éstas.

Variables locales, de pila y de excepción

Las instrucciones o bytecodes que el intérprete es capaz de procesar pueden utilizar tres tipos de variables que, aunque se diferencian explícitamente en las definiciones de las citadas instrucciones, el intérprete procesa de manera idéntica.

Los tres tipos de variables son:

- variables locales; cuyo nombre sigue la forma 'l(n)' donde n es un valor entero positivo y

cuyo valor debe corresponderse con uno de los tipos de datos válidos definidos en el sistema, con una localización de memoria váida o con el valor nulo.

- variables de pila; cuyo nombre sigue la forma 's(n)' donde n es un valor entero positivo y cuyo valor debe corresponderse con uno de los tipos de datos válidos definidos en el sistema, una localización de memoria valida o con el valor nulo.

- variables de excepción; cuyo nombre sigue la forma 'e(n)' donde n es un valor entero positivo y cuyo valor debe corresponderse con una localización de memoria valida o con el valor nulo.

Heap

Los dos tipos de estructuras dinámicas con las que el intérprete va a ser capaz de lidiar son los objetos y los arrays. De nuevo, ambos van a ser manipulados de manera idéntica por el intérprete aunque su concepto sea muy diferente.

Tanto los arrays como los objetos van a almacenarse en una estructura que simula un heap y que el intérprete utilizará para la creación, consulta y manipulación de éstos.

Se ha asumido que el heap está dividido en localizaciones de tamaño variable y que cada array u objeto almacenado en él ocupa única y exclusivamente una localización. Dicha localización se representa con un entero positivo.

En las secciones *Arrays: creación, acceso y manipulación*, y *Objetos: creación, acceso y manipulación* se detalla la implementación de ambos de una manera más profunda.

Organización de la memoria

La organización de la memoria que se ha establecido para el correcto mantenimiento del estado del intérprete utiliza tres predicados:

- *table/3*
  Cuyo desglose es: table(nombre-bloque, nombre-variable, valor-variable)
  Se utilizará para representar una variable local o de pila. Por lo tanto existirán tantos predicados *table/3* como variables locales o de pila contenga el bloque que en ese momento esté tratanto el intérprete.
  'nombre-bloque' debe corresponderse con un nombre de bloque válido contenido en la rbr.
  'nombre-variable' y 'valor-variable' deben cumplir los requisitos especificados en la sección *Variables locales y de pila.*

- *heap/2*
  Cuyo desglose es: heap(localizacion, tipo)
  Se utilizará para representar un objeto o array creado durante la ejecución. Existirán tantos predicados *table/3* como objetos o arrays.

'localizacion' debe corresponderse con un entero positivo único.
'tipo' debe ser un tipo de datos válido que, en el caso de los objetos se corresponderá con el tipo de éstos y, en el caso de los arrays, se corresponderá con el tipo de las componentes de éstos.

- *attribute/3*
  Cuyo desglose es: attribute(localizacion, nombre-atributo, valor-atributo)
  Se utilizará para representar, en el caso de los objetos, los atributos contenidos en éstos y, en el caso de los arrays, el tamaño y el valor de las componentes de éstos.
  En el caso de los objetos, existirán tantos predicados *attribute/3* como atributos contenga el objeto.
  En el caso de los arrays, existirán tantos predicados *attribute/3* como componentes tenga el array más un predicado *attribute/3* adicional que almacenará el tamaño del array.

Estructura de los marcos o frames

Como ya se ha comentado anteriormente, cada bloque contenido en la rbr tiene asociado un marco o frame único independiente de todos los demás.

Para la representación de dicho marco se utilizarán un conjunto de predicados *table/3*, tantos como variables locales o de pila tenga especificado el bloque en su definición.
Todos los predicados *table/3* asociados a un bloque tendrán en común el valor de 'nombre-bloque' que se corresponderá con el nombre único del bloque.

Tomando como ejemplo la definición de este bloque en la rbr:

```
12([l(0), s(1), s(2)],[s(1)],[e(1)],[A]) :=
    assr(cmp(icmpgt), [s(1), s(2)], [])
    call(block, 6, [[l(0)], [s(1)], [e(1)], [A]], [])
```

tendremos los siguientes predicados:

```
table(12, l(0), valor-de-l(0))
table(12, s(1), valor-de-s(1))
table(12, s(2), valor-de-s(2))
```

Tipos de datos básicos

Los tipos de datos básicos implementados en el intérprete son los siguientes:

- I: se corresponde con el tipo 'integer' de Java. Ocupa una única variable local o de pila.
- Z: se corresponde con el tipo 'boolean' de Java. Ocupa una única variable local o de pila.
- F: se corresponde con el tipo 'float' de Java. Ocupa una única variable local o de pila.
- C: se corresponde con el tipo 'char' de Java. Ocupa una única variable local o de pila.
- D: se corresponde con el tipo 'double' de Java. Ocupa dos variable locales o de pila. (*1)El mismo valor se guarda tanto en la primera variable como en la segunda.

- J: se corresponde con el tipo 'long' de Java. Ocupa dos variable locales o de pila. *El valor se guarda en la primera variable mientras que en la segunda se almacena un 0.

(*1)Esto es así porque aunque en la representación recursiva el tipo de datos se representa mediante dos variables locales o de pila, en el intérprete solo es necesario que el valor se encuentre en una de ellas.

Procesado de las aserciones

Cada una de las aserciones explicadas en la subsección *Los bytecodes de la representación RBR* de la sección *Representación recursiva basada en reglas (RBR)* tiene asociada un precidado step/2 que interpreta y procesa su función.

El desglose de step es el siguiente: step(definicion-guarda, nombre-bloque).

- 'definicion-guarda' sigue el esquema assr(tipo-assr,variables-entrada,variables-salida) explicado en la subsección citada anteriormente.
- 'nombre-bloque' se corresponde con el nombre del bloque donde está contenida la instrucción assr y es necesario para que se pueda acceder a las variables locales y de pila por medio del predicado *table/3*.

El siguiente ejemplo muestra el precicado step asociado a un assr no destructivo:

```
step(assr(int_nonzero,[InArg], _OutArgs), RuleHead) :- !,
   table(RuleHead, InArg, Value),
   Value \= 0.
```

Procesado de los bytecodes

Cada uno de los bytecodes explicados en la subsección *Los bytecodes de la representación RBR* de la sección *Representación recursiva basada en reglas (RBR)* tiene asociada un precidado step/2 que interpreta y procesa su función.

El desglose de step es el siguiente: step(definicion-guarda, nombre-bloque).

- 'definicion-guarda' sigue el esquema assr(tipo-assr,variables-entrada,variables-salida) explicado en la subsección citada anteriormente.
- 'nombre-bloque' se corresponde con el nombre del bloque donde está contenida la instrucción assr y es necesario para que se pueda acceder a las variables locales y de pila por medio del predicado *table/3*.

Llamadas a bloques

En esta sección se explica en profundidad cómo se ha llevado a cabo lo comentado subsección *Paso y retorno de variables. El bytecode call.* de la sección *Representación recursiva basada en reglas (RBR)* sobre llamadas entre bloques.

El predicado que realiza el procesado del bytecode call es el siguiente:

```
step_method_execution(call(_Type, NewRuleHead,[InArgs, OutArgs, Exceptions, CallState],_), RuleHead) :-
   rbr_rule(NewRuleHead, [InVarsBlock, OutVarsBlock, OutExceptionsBlock, _], _Bytecodes, _),!,
   frame_fill_exceptions(InArgs,RuleHead),
   frame_create(RuleHead, NewRuleHead, InArgs, InVarsBlock),
   frame_backup(RuleHead, BackupVars),
   frame_destroy(RuleHead),
   (stepRule(NewRuleHead,CallState) ; true),
   frame_restore(RuleHead, BackupVars),
   frame_update(RuleHead, NewRuleHead, OutArgs, OutVarsBlock, Exceptions,  OutExceptionsBlock),
   frame_destroy(NewRuleHead).
```

Lo primero a tener en cuenta es que en la definición del predicado *step_method_execution/2*, se instancian en *InArgs* las variables de entrada pertenecientes al bloque anterior que se tienen que copiar en el bloque siguiente y en *OutArgs* las variables de salida pertenecientes al bloque siguiente que se tienen que copiar en el bloque anterior. Los nombres de ambas listas se corresponden con variables del bloque anterior.

Por otra parte, la primera instrucción del cuerpo del predicado,

```
rbr_rule(NewRuleHead, [InVarsBlock, OutVarsBlock, OutExceptionsBlock, _], _Bytecodes, _)
```

Instancia en *InVarsBlock* y *OutVarsBlock*, los nombres de las variables de entrada y salida del bloque siguiente, respectivamente.

Ambas listas de variables de entrada y de variables de salida del bloque anterior y del bloque siguiente deben corresponderse en tamaño dos a dos. En caso de que esto no se cumpla habrá un error en la RBR y el intérprete no podrá continuar.

La siguiente instrucción,

```
frame_fill_exceptions(InArgs,RuleHead)
```

realiza el proceso de instanciar en la memoria, por medio de predicados *table/3,* las variables de excepción que puedan necesitarse y aun no estar definidas por encontrarse la máquina en un estado *normal*.

Mediante el siguiente grupo de instrucciones,

```
frame_create(RuleHead, NewRuleHead, InArgs, InVarsBlock)
frame_backup(RuleHead, BackupVars)
frame_destroy(RuleHead)
```

se crea un nuevo marco para el bloque siguiente con las variables de entrada copiadas y renombradas utilizando la lista de variables de *InArgs* y los nombres de variables de entrada de

InVarsBlock, se hace una copia de las variables del bloque anterior y se elimina el marco del bloque anterior.

La siguiente intrucción hace la llamada al bloque siguiente propiamente dicha:

```
(stepRule(NewRuleHead,CallState) ; true)
```

en ella es importante destacar que siempre se ejecutará con éxito. El motivo de esto es que, a partir de esa llamada, puede darse el caso de que todas las guardas que hacen que sea posible saltar a otros bloques no se ejecuten con éxito y sea necesaria la vuelta atrás para continuar con la ejecución justo a la vuelta de la llamada incial. La función del *true* resaltado es precisamente ésta.

Por último, las instrucciones restantes:

```
frame_restore(RuleHead, BackupVars)
frame_update(RuleHead, NewRuleHead, OutArgs, OutVarsBlock, Exceptions,  OutExceptionsBlock)
frame_destroy(NewRuleHead)
```

restauran el marco del bloque anterior con las variables sin actualizar de éste, actualizan dichas variables teniendo en cuenta el renombramiento que debe producirse entre variables de salida del bloque siguiente y las variables del bloque anterior y eliminar el marco del bloque siguiente.

Métodos nativos

Aunque la potencia de la API de Java es más que suficiente para casi todo tipo de aplicaciones, algunas de ellas necesitan utilizar la potencia específica de una plataforma en concreto, por ejemplo para conseguir un poco más de velocidad y eficiencia.

Java permite incorporar en sus programas fragmentos de código nativo es decir, código compilado para una determinada plataforma, generalmente escrito en C/C++. Así se puede utilizar código específico de una plataforma, bibliotecas ya escritas...

Esto hace que los métodos nativos de los que haga uso el código de bytes inicial no se encuentren definidos en la rbr y sea necesario una reimplementación en Prolog en el propio intérprete.

Como ya se mencionó en la sección *Esquema del código fuente* los métodos nativos reimplementados se encuentran en el archivo profiler_native_methods.pl.

Como ejemplo, alguno de los métodos nativos cuya reimplementación ha sido necesaria son:

'java/io/PrintStream_println(Ljava/lang/String;)V', utilizado para poder escribir por la consola medienta System.out.println().

'java/lang/String_equals(Ljava/lang/Object;)Z', utilizado para realizar comparaciones de Strings.

'java/lang/String_length()I', utilizado para contabilizar la longitud de un String.

Creación en memoria de los argumentos de entrada

Una vez realizada la llamada al sistema, ya sea desde la interfaz web o desde una consola, y en caso de que se haya especificado el paso de parámetros de entrada, la creación en memoria de éstos se realiza por medio del predicado *init_in_args/3*.

Por cada argumento de entrada se asignará una variable local (excepto en el caso de los tipos long y double que se asignarán dos) que contendrá el valor del argumento si el tipo de éste es simple o una referencia al heap si el tipo es String o array de String. El predicado asignará la variable local l(0) al primer argumento y, a partir de ésta primera, las siguientes de manera secuencial.

Como se ha comentando anteriormente, para el almacenamiento de las variables locales se utilizarán tantos predicados *table/3* como sean necesarios teniendo todos ellos el campo 'nombre-bloque' común con el nombre del método inicial.

En función del tipo del argumento de entrada lo presentado en memoria es lo siguiente:

- Tipos integer, boolean y float: un predicado *table(nombre-bloque, nombre-variable, valor-variable)* donde 'nombre-bloque' se corresponderá con el nombre del bloque inicial, 'nombre-varibale' con una variable local y 'valor-variable' con el valor del argumento de entrada.
- Tipos long y double: dos hechos *table(nombre-bloque, nombre-variable, valor-variable)* y *table(nombre-bloque, nombre-variable2, valor-variable2)* donde 'nombre-bloque' se corresponderá con el nombre del bloque inicial, 'nombre-variable1' con una variable local, 'nombre-variable2' con al siguiente variable local, 'valor-variable1' con el valor del argumento de entrada y 'valor-variable2' con el valor 0.

  *Esto es así porque aunque en la representación recursiva el tipo de datos se representa mediante dos variables locales o de pila, en el intérprete solo es necesario que el valor se encuentre en una de ellas.
- Tipo String: un hecho *table(nombre-bloque, nombre-variable, valor-variable)* donde 'nombre-bloque' se corresponderá con el nombre del bloque inicial, 'nombre-variable' con una variable local y 'valor-variable' con la localización del heap donde se ha creado el objeto String mediante el uso del predicado *create_string/2*.
- Tipo array de String: un hecho *table(nombre-bloque, nombre-variable, valor-variable)* donde 'nombre-bloque' se corresponderá con el nombre del bloque inicial, 'nombre-variable' con una variable local y 'valor-variable' con la localización del heap donde se ha creado el Array de String mediante el uso del predicado *create_string_array/3*.

Inicialización del intérprete

Una vez que el script *profiler* hace la llamada al predicado *costa_profiler/1,* se inicializa el intérprete, se tiene en cuenta si el método de entrada es estático o no para establecer la primera variable local del bloque principal, se crean los argumentos de entrada y se llama al sistema COSTA para que transforme el código de bytes inicial en la RBR utilizada durante el análisis.

La inicialización del intérprete consiste en asegurarse de que la memoria del sistema comienza estando vacía. Para ello elimina de la base de datos todos los hechos para el manejo de la memoria: *table/3*, *heap/2*, *attribute/3*; todos los hechos para el manejo de campos estáticos: *static_field/3, userc_field/5, static_initialized/1;* los hechos para el realizar el análisis: *methods_executed/2, news_executed/2*, junto con el reseteo del contador de instrucciones.

Si el método inicial no es *static* quiere decir que necesita de la creación de un objeto para poder ejecutarse y este objeto debe guardar una referencia al objeto padre que lo creó. Por ello, el intérprete crea una variable local l(0) en el bloque inicial cuyo valor es el nombre de dicho bloque. Esta variable, aunque no se utiliza, es necesaria para mantener la correspondencia en la RBR.

Arrays: creación, acceso y manipulación

La información que va a estar relacionada con cada array va a ser:

1. localización del heap en la que se encuentra.
2. tipo de datos de los elementos del array.
3. Tamaño.
4. pares del tipo 'indice' - 'valor' donde 'indice' es un entero positivo y 'valor' se corresponde con uno de los tipos de datos válidos definidos en el sistema, una localización de memoria valida o con el valor nulo. Cada uno de estos pares se corresponderá con un índice del array.

Nuestro sistema divide el conjunto de los arrays en tres grupos con un comportamiento distinto en función del tipo de datos y las dimensiones de los mismos.
Dichos tres grupos son:

- Arrays de una dimensión cuyos elementos son de tipo básico.
- Arrays de una dimensión cuyos elementos son objetos.
- Arrays multidimensionales.

El proceso de creación de arrays es el siguiente:

Primero se busca el primer hueco libre en el heap mediante el predicado “last_heap_location(X)”, que devuelve en X la primera posición libre en el heap, para colocar el array que va a ser creado en dicho hueco. Tras ello se guarda el tipo de los elementos del array en dicha localización mediante la asercion “asserta(heap(localizacion,tipo))” siendo “localización” la primera posición libre en el heap devuelta por “last_heap_location/1” y “tipo” el tipo de los elementos del array. Despues de esto se guarda el tamaño del array mediante la aserción “asserta(attribute(loc,size,tamaño))” siendo “loc” la localización asignada al array, “size” una constante para indicar que el atributo que se esta guardando es el tamaño del array y “tamaño” el tamaño del array.
Una vez guardado el tamaño del array, solo queda inicializar sus componentes, lo que se hará con una aserción del tipo “asserta(attribute(loc,indice,valor))” para cada componente del mismo, siendo “loc” la localización del array, “indice” el indice que indica el número de componente del array y “valor” el valor de dicha componente para su inicialización (0 para tipos básicos, null para objetos).

Así pues en función del tipo de arrays tras la creación de los mismos nos encontraremos con las siguientes estructuras:

Arrays cuyos elementos son de tipo básico

Suponemos un array de enteros de 4 elementos y la posición 8 del heap como primera posición libre.

```
heap(8,primitiveType(int))
attribute(8,size,4)
attribute(8,0,0)
attribute(8,1,0)
attribute(8,2,0)
attribute(8,3,0)
```

Arrays cuyos elementos son objetos

Suponemos un array de Strings de 5 elementos y la posición 5 del heap como primera posición libre.

```
heap(5,'java/lang/String')
attribute(5,size,5)
attribute(5,0,null)
attribute(5,1,null)
attribute(5,2,null)
attribute(5,3,null)
attribute(5,4,null)
```

Arrays multidimensionales

Suponemos un array de int de 2 dimensiones de 3 y 4 elementos respectivamente y la posición 7 del heap como primera posición libre.

Para el array principal tenemos:

```
attribute(7,size,3)
attribute(7,0,8)
attribute(7,1,9)
attribute(7,2,10)
```

Vemos que en este caso no es necesario guardar el tipo de los elementos.

Para los elementos del array:

```
heap(8,'primitiveType(int)')
attribute(8,size,4)
attribute(8,0,0)
attribute(8,1,0)
attribute(8,2,0)
attribute(8,3,0)

heap(9,'primitiveType(int)')
attribute(9,size,4)
attribute(9,0,0)
```

```
attribute(9,1,0)
attribute(9,2,0)
attribute(9,3,0)

heap(10,'primitiveType(int)')
attribute(10,size,4)
attribute(10,0,0)
attribute(10,1,0)
attribute(10,2,0)
attribute(10,3,0)
```

Veamos como se asignarian los elementos del array en función de los tres grupos.

Arrays cuyos elementos son de tipo básico

Es la asignación más sencilla de los tres grupos. Simplemente se rellenan los campos attribute(loc,indice,valor) con los valores correspondientes.

Suponiendo el array de enteros del ejemplo anterior relleno con los siguientes datos [2,4,6,8] tendríamos la siguiente estructura:

```
heap(8,primitiveType(int))
attribute(8,size,4)
attribute(8,0,2)
attribute(8,1,4)
attribute(8,2,6)
attribute(8,3,8)
```

Arrays cuyos elementos son objetos

En este tipo de arrays hay que tener en cuenta que lo que se va a meter en el campo "valor" de attribute(loc,indice,valor) es un número entero que indicará la posición del heap donde se encuentra el objeto.

Suponiendo el array de Strings del ejemplo anterior relleno con los siguientes datos ["dato1","dato2","dato3","dato4","dato5"] y suponiendo que la nueva primera posición vacía del heap es la número 6, tendríamos la siguiente estructura:

```
heap(5,'java/lang/String')
attribute(5,size,5)
attribute(5,0,6)
attribute(5,1,8)
attribute(5,2,10)
attribute(5,3,12)
attribute(5,4,14)
```

Nótese que las posiciones de los Strings en el heap no son contiguas. Esto se debe a que los Strings, como veremos en la sección *El tipo String,* son tratados como arrays de caracteres y por lo tanto requieren 2 posiciones del heap.

Arrays multidimensionales

Al igual que en el tipo anterior, en este tipo de arrays hay que tener en cuenta que lo que se va a meter en el campo "valor" de attribute(loc,indice,valor) de todas las dimensiones del array,

excepto de la última, es un número entero que indicará la posición del heap donde se encuentran los elemento de tipo array.
Además hay que tener en cuenta que los arrays se crean en profundidad, por lo que en el orden de asignación del heap, puede que dos elementos consecutivos dela misma dimensión estén bastante separados en el heap.

Para remarcar el orden de asignación de heap vamos a suponer un array de int de tres dimensiones de 2 elementos cada una relleno con los siguientes datos [[[“1”,”2”],[“3”,”4”]], [[“5”,”6”],[“7”,”8”]]] tendríamos la siguiente estructura:

Para la primera dimensión del array tenemos:

```
attribute(7,size,2)
attribute(7,0,8)
attribute(7,1,11)
```

Para la segunda dimensión del array:

```
attribute(8,size,2)
attribute(8,0,9)
attribute(8,1,10)

attribute(11,size,2)
attribute(11,0,12)
attribute(11,1,13)
```

Para la última dimensión:

```
heap(9,'java/lang/String')
attribute(9,size,2)
attribute(9,0,loc(String))
attribute(9,1,loc(String))
```

siendo loc(String) la localización que le sea asignada al String, que dependerá del orden de asignación (por ejemplo podría asignarse antes int[1][1][1], que int[0][0][0] y tener un número de heap menor). Si fueran elementos de tipo básico, simplemente se almacenaría el valor.
La misma estructura que para el elemento 9 la tendríamos para las localizaciones 10, 12 y 13.

Como se puede observar, debido a la creación en profundidad de los arrays en la primera dimensión los números de heap de los 2 elementos difieren en 3 posiciones. Esto se acentuará más cuantas más dimensiones tenga el array.

Las reglas de nuestro sistema que se identifican con la creación de arrays según los tres grupos que hemos diferenciado son:

- *step(bytecode(_, newarray(Type),[ArraySizeVar],[ArrayLocationVar],_),RuleHead)* para arrays de tipo básico.
- *step(bytecode(_, anewarray(ObjectType),[ArraySizeVar], [ArrayLocationVar],_),RuleHead)* para arrays de objetos.
- *step(bytecode(_,multianewarray(_,_)[FirstSize|RestSizes], [ArrayLocationVar],_),RuleHead)* para arrays multidimensionales.

Las tres reglas se encuentran en el archivo "costa_profiler.pl".
Además de estas reglas que dirigen de forma general la creación de arrays en el archivo "profiler_arrays.pl" hay funciones de apoyo para la creación y manejo de arrays.
Son las siguientes:

- *init_array_components/4*: inicia al valor por defecto los elementos de un array.
- *init_multiarray_components/7*: inicia los elementos de una dimensión de un multiarray que no sea la última.
- init_multiarray/4: función que decide en función de la dimensión de la que se trate si llamar a "init_multiarray_components" para crear una dimensión intermedia del multiarray, o a "init_array_components" para crear la última.

Así pues la forma de crear los arrays desde el punto de vista del código y en función del tipo de array que se cree es:

Arrays multidimensionales

Primero nos encontraremos con la regla: *step(bytecode(_,multianewarray(_,_)[FirstSize| RestSizes], [ArrayLocationVar],_),RuleHead)*, que llamará a *init_multiarray/4*, que en función de si es la última dimensión o no del array llamara respectivamente a *init_array_components/4* o *init_multiarray_components/7*.

Arrays cuyos elementos son objetos

Primero nos encontraremos con la regla: *step(bytecode(_, anewarray(ObjectType), [ArraySizeVar],[ArrayLocationVar],_),RuleHead)*, que llamará a *init_array_components/4.*

Arrays cuyos elementos son de tipo basico

Primero nos encontraremos con la regla,

step(bytecode(_, newarray(Type),[ArraySizeVar],[ArrayLocationVar],_),RuleHead),

que llamará a *init_array_components/4*.

El tipo String

Java representa los Strings como arrays de caracteres. Debido a esto la representación de las variables de tipo String en nuestro sistema va a estar formada por la siguiente información:

1. Localización del heap en la que se encuentra.
2. Tamaño del String.
3. Offset.
4. Array de char que forma el String.

El campo Offset se utilizará para operaciones sobre Strings en las que haya que tener en cuenta un índice a partir del cual se realizarán las mismas, y representa dicho índice. Un ejemplo de este tipo de operaciones es el método substring(int X) de la clase String, que devuelve una subcadena de la cadena original a partir del elemento X (posición X del array de char que forma el String).

La regla de nuestro sistema que trata la creación de Strings es la siguiente:

```
step(bytecode(_, push(ldc, string(utf8(Value))), _InArgs, [OutArg], _), RuleHead) :- !,
                increase_instructions,
                create_string(Value,Location),
                retractall(table(RuleHead,OutArg,_)),
                asserta(table(RuleHead, OutArg, Location)).
```

Cuando se defina una variable de tipo String, en la RBR nos encontraremos con una regla de la forma *step(bytecode(_, push(ldc, string(utf8(Value))), _InArgs, [OutArg],), RuleHead),* siendo "Value" el String en cuestión. Una vez nos encontremos con esta regla llamaremos a la regla *createstring(Value,Location)* que se encargará de crear el String "Value".

La definición de esta regla en nuestro sistema es la siguiente:

```
create_string(Value,StringLocation) :- !,
            get_heap_location(StringLocation),
            asserta(heap(StringLocation, 'java/lang/String')),
            atom_chars(Value,CharList),
            length(CharList, Count),
            asserta(attribute(StringLocation, 'count:I', Count)),
            asserta(attribute(StringLocation, 'offset:I', 0)),
            get_heap_location(CharArrayLocation),
            asserta(heap(CharArrayLocation,'primitiveType(char)')),
            create_char_array(CharArrayLocation, CharList, 0),
            asserta(attribute(StringLocation, 'value:[C', CharArrayLocation)).
```

Lo primero que hace es buscar la primera localización libre del heap mediante *get_heap_location(StringLocation)*. Tras esto almacena en la base de datos el tipo (java/lang/String) en dicha localización mediante la aserción *asserta(heap(StringLocation, 'java/lang/String'))*, para indicar que en dicha posición del heap hay un String. Despues mediante *atom_chars(Value,CharList)* transforma el String de entrada "Value" en una lista "CharList" que contiene el array de chars que forma el String. Calcula su longitud mediante *length(CharList,Count)* y almacena tanto esta longitud, como la inicialización del offset a 0, mediante las dos siguientes aserciones: *asserta(attribute(StringLocation, 'count:I', Count))* y *asserta(attribute(StringLocation,'offset:I',0))*. En estas aserciones "StringLocation" es la posición del String en el heap y "Count" la longitud de éste, mientras que 'count:I' y 'offset:I' son constantes para indicar que lo qué se almacena en esos atributos son la longitud y el offset del String respectivamente. Tras esto, se crea e inicializa el array de chars que forma el String. Para ello se utiliza el mismo procedimiento utilizado en la creación de Arrays cuyos elementos son de tipo primitivo explicado anteriórmente: se busca una posición libre del heap, se aserta *heap(CharArrayLocation,'primitiveType(char)')* siendo "CharArrayLocation" esa posición y se asertan todos los pares atributo-valor correspondientes a los caracteres mediante la regla *create_char_array(CharArrayLocation,CharList,0)*.

Tras esto sólo falta guardar en la base de datos la posición donde se ha almacenado el array de

caracteres correspondiente al String, lo que se hace mediante la aserción *"asserta(attribute(StringLocation,'value:[C',CharArrayLocation))",* donde "StringLocation" es la posición del heap donde está almacenado el String, "CharArrayLocation" es la posición donde está almacenado el array de char y 'value:[C' es una constante para indicar que estamos guardando la posición del array de char.

De esta forma, suponiendo que se define el String "Prueba" y que la primera posición libre del heap es la posición 5, tras la definición del String tendríamos las siguientes estructuras:

Para el String:

```
heap(5,'java/lang/String')
attribute(5,'count:I',6)
attribute(5,'offset:I',0)
attribute(5,'value:[C',6)
```

Para el Array de char:

```
heap(6,'primitiveType(Char)')
attribute(8,size,6)
attribute(8,0,X0)
attribute(8,1,X1)
attribute(8,2,X2)
attribute(8,0,X3)
attribute(8,1,X4)
attribute(8,2,X5)
```

X_0, X_1, X_2, X_3, X_4 y X_5, son números enteros que se corresponden con los caracteres 'P','r','u','e','b','a', ya que no se almacenan los caracteres, sino una representación numérica de los mismos que se halla con la función char_code(Char, Int), que devuelve en "Int" el código numérico para el char "Char".

Además de las reglas comentadas, se han definido en el archivo *profiler_arrays.pl* las siguientes reglas para ayudar a la gestión de Strings:

- array_char_print/3 para imprimir Strings
- copy_array/6 para copiar un array en otro
- compare_array_char/4 para comparar arrays de char

Objetos: creación, acceso y manipulación

En el caso de los objetos, la información que cada uno de ellos debe contener:

1. localización en el heap donde se encuentra.
2. tipo del objeto.
3. pares del tipo 'atributo' - 'valor' donde 'atributo' es un nombre de atributo válido y 'valor' se corresponde con uno de los tipos de datos válidos definidos en el sistema, una localización de memoria valida o con el valor nulo. Cada uno de estos pares se corresponderá con un atributo del objeto.

La regla de nuestro sistema que crea un objeto es la siguiente:

```
step(bytecode(_, new(ObjectType), _InArgs, [OutArg], _), RuleHead) :- !,
                print_memory,
                increase_instructions,
                increase_news(ObjectType),
                get_heap_location(I),
                asserta(heap(I, ObjectType)),
                init_fields(ObjectType,I),
                init_inherited_fields(ObjectType,I),
                retractall(table(RuleHead,OutArg,_)),
                asserta(table(RuleHead, OutArg, I)).
```

Su funcionamiento se detalla a continuación.

Primero se busca un hueco libre en el heap mediante "get_heap_location(I)", que devuelve en I la primera posición libre del mismo. Una vez se tiene la posición donde se va a almacenar el objeto, se almacena su tipo mediante la siguiente aserción "asserta(heap(I,ObjectType)), siendo "I" la localización libre que nos ha devuelto la función "get_heap_location" y "ObjectType" el tipo del objeto en cuestión. Tras eso lo que se hace es iniciar sus atributos mediante "init_fields(ObjectType,I)" y los atributos heredados de sus clases padre mediante "init_inherited_fields(ObjectType,I)".

La regla "init_fields(ObjectType,I)" que inicia sus atributos es de la siguiente forma:

```
init_fields(ObjectName,HeapLocation):-
                findall((FieldName,FieldType),
                jbcr_field(ObjectName,FieldName,_,FieldType,_,_),FieldsList),
                static_initializer(ObjectName,_RuleHead),
                init_fields_aux(HeapLocation, FieldsList)
```

Su funcionamiento es el siguiente:

Primero se buscan todos los atributos que se han generado tras la ejecución del programa (recordemos que antes de ejecutar el profiler el programa java ha sido ejecutado, gracias a lo que podemos tener este tipo de información) con sus tipos mediante la regla findall((FieldName,FieldType).

Tras esto, mediante la regla jbcr_field(ObjectName,FieldName,_,FieldType,_,_),FieldsList), obtenemos los atributos que pertenecen al objeto que estamos creando. Mediante el predicado *static_initializer(ObjectName,_RuleHead)*, iniciamos los posibles atributos estaticos del objeto en caso de que los hubiera. Este proceso se explica más detalladamente en la sección posterior *Campos estáticos: inicialización y acceso*. Tras esto llamamos a la regla init_fields_aux(HeapLocation, FieldsList), que iniciará la lista de atributos del objeto a sus valores por defecto.

Esta última regla se llama de manera recursiva para iniciar todos los atributos del objeto.

Para iniciar los atributos heredados de las clases padre se llama a la regla "init_inherited_fields(ObjectType,I)" que tiene la siguiente forma:

```
init_inherited_fields(ObjectName,HeapLocation):-
        findall((ClassType),
        jbcr_sub_class(ObjectName,ClassType),ClassTypes),
        init_inherited_fields_aux(HeapLocation, ClassTypes).
```

Esta regla busca todas las clases que se generan, tras la ejecución del programa java, mediante "findall(ClassType)" y consulta cuales de esas clases son clases padres de la clase del objeto que se está creando mediante "jbcr_sub_class(ObjectName,ClassType),ClassTypes)". Esta regla devuelve en ClassTypes la lista de las clases padres del objeto que se está creando. Una vez tenemos esa lista lo único que hay que hacer es llamar a la regla "init_inherited_fields_aux(HeapLocation, ClassTypes)", que se llamará recursivamente para inicializar todos los campos heredados del objeto a sus valores por defecto.

Supongamos una clase "prueba" con 2 argumentos "campo_entero" de tipo int y "campo_string" de tipo String, que hereda de una clase "padre" con un argumento "campo_entero_padre" de tipo int y como primera posición libre del heap la posición número 9.

Tras crearse un objeto de la clase "prueba" tendremos la siguiente estructura:

```
heap(9,"prueba")
attribute(9,"campo_entero",0)
attribute(9,"campo_string",null)
attribute(9,"campo_entero_padre",0)
```

En el archivo "costa_profiler.pl" hay otras dos reglas referentes al manejo de objetos y clases que son las siguientes:

- *step(bytecode(_, instanceof(ObjectType), [LocationVar], [OutArg], _), RuleHead)*, que implementa la funcion instanceof de java, es decir devuelve cierto si un objeto dado es miembro de una clase dada o falso en caso contrario. Para ello busca en la posición del heap donde está almacenado el objeto y comprueba su tipo para ver si coincide con el ObjectType presente en la regla.
- *step(bytecode(_, checkcast(ObjectType), [LocationVar], [_OutArg], _), RuleHead),* que comprueba la validez de un casting hecho a un objeto. Para ello actua igual que la regla anterior, buscando en la posición del heap que ocupa el objeto, para comparar su tipo.

Campos estáticos: inicialización y acceso

Partiendo de la implementeación de los atributos no estáticos con herencia, hubo que abordar la implementación de los estáticos. Para ello, se creó una estructura donde guardar para cada clase sus atributos estáticos.

Para tratar la herencia se asciende en la jerarquía de clases hasta llegar a la clase donde se definió el atributo, al igual que ocurría con los atributos no estáticos.

Además de tener en cuenta que los atributos estáticos son los mismos para todos los objetos que

sean instancia de una clase, hay que utilizar inicializadores para éstos. Durante el desarrollo del proyecto el sistema COSA no incluía en la RBR bloques que se correspondieran con los inicializadores de campos estáticos, por lo que este asunto está todavía sin terminar.

En java existen bloques de inicialización de estáticos, que pueden contener toda clase de instrucciones. Se declaran así:

```
static
{
//instrucciones que se ejecutan al inicializar lo estáticos, una sola vez
}
```

Las asignaciones que se llevan a cabo al declarar un atributo estático formarán parte de este método. De forma transparente se introducen al principio del mismo cuando se crea la clase (en tiempo de compilación del .class).

El método especial *static* de una clase se debe inicializar una sola vez, inmediatamente antes de en los siguientes casos:

- Cuando se accede a un atributo estático (que no sea final, ya que éstos son constantes y no interfieren) en una instancia de la clase, lo que engloba a las herederas.
- Cuando se invoca un método estático que pertenece a una instancia de la clase.
- Cuando se crea un objeto que es instancia de la clase.

Como se puede ver, durante la creación de un objeto, la inicilazación de los atributos estáticos de una clase se debe hacer inmediatamente antes de la inicialización de los atributos no estáticos (estos se inicializan a su valor por defecto). Por ello se ha incluido al hacer una operación de *new (*en concreto dentro de la claúsula *init_fields)* la comprobación y, si es necesaria, la ejecución del inicializador cuando se recorre la jerarquía de clases para inicializar el objeto.

Para la posible implementación futura de los inicializadores, siempre que se dé el caso de una de las situaciones anteriores, se deberá comprobar la aserción Prolog ***static_initialized(****Clase****)*** que comprueba si para 'Clase' se ha llevado a cabo la invocación al inicializador y, si no ha sido así, se llama al predicado ***static_initializer(****Clase, Identificador****)*** que comprueba de nuevo si está inicializada la clase, teniendo en cuenta que deberá ejecutar las instrucciones del inicializador cuando se disponga del mismo.

La variable 'Identificador' se ha dejado en desuso y podría servir para pasar el identificador de una regla.

Siempre que se dé alguno de los sucesos que disparan la ejecución de los inicializadores se recorre la jerarquía de clases, haciendo lo siguiente:

1. Se llega hasta la clase Object (ascensión por la jerarquía).
2. Se ejecuta su inicializador estático.
3. A continuación se desciende en la jerarquía de clases ejecutando los inicializadores estáticos para cada uno de los niveles hasta llegar al objeto que se quería inicializar.

Excepciones

La implementación de excepciones requiere de tres partes:

Implementación de los bytecodes de excepción

1. **ie_throw**(*Excepcion*), implementada igual que la instrucción **new**(*Excepcion*).
2. athrow y assign, gracias al control mediante la unificación y el uso de guardas, se comportan igual, copiando el contenido de una variable de entrada a una de salida.

Modificación del paso de parámetros para que contemplen las variables de tipo e(N)

En una primera aproximación al problema de implementar el profiler no se copiaban las variables de excepción e(N) por lo que más adelante hubo que integrarlas en el sistema de paso de parámetros.

Si no se producen excepciones las variables de excepción no contienen nada y es necesario rellenarlas con null. Esto se hace durante la actualización del marco o frame durante la transición de un bloque a otro.

Para ello se hace uso del predicado:

frame_fill_exceptions(*lista-variables, nombre-bloque****).***

que, dada una lista de variables de excepción 'lista-variables', comprobará si cada una de ellas se encuentran en el bloque actual 'nombre-bloque'. Por cada una que no encuentre asertará un hecho *table/3* con valor null para la excepción correspondiente.

Unificación de las variables de estado de la ejecución

Cuando se procesa un bytecode call, las variables que controlan el estado de la máquina pueden estar o no instanciadas (si lo están será a uno de los valores *normal* o *exception*). Si la ejecución de la regla tiene éxito (las guardas no causan el fallo porque son ciertas), es necesario que éstas variables del estado de la maquina queden unificadas con las variables que tiene el bloque llamado.

Para entender mejor cómo se debe comportar la propagación propondré un ejemplo muy sencillo.

Dada la clase:

```
package paquete;

class wildcard {

   public static int main(String args[]){
      return 6/0;
   }
}
```

Tras mostrar los bloques generados se hará una traza de la secuencia de llamadas a los bloques:

En el bloque 5 hay una guarda que comprueba si el divisor es cero. Esta comprobación se hace

siempre que haya una división en la que el operando pueda ser cero. Si se divide entre un operador explícito distinto de cero no se genera.

Como se puede ver, la instrucción assr fallará porque el operador sí es cero y, entonces, mediante backtraking, se probará la segunda regla cuyo identificador es 5. Aquí se creará un objeto de tipo *java.lang.ArithmeticException* y se llamará al bloque 2. En éste la variable que monitoriza el estado de la ejecución, **A**, se unificará con *exception*. Esta variable queda unificada, lo que conlleva que a partir de ahí, se tendrá en cuenta esta unificación para los siguientes encajes.

Si en vez de divivir por cero hubieramos usado otra variable con valor distinto, se hubiera podido seguir el camino por el bloque 3, en cuyo caso **A** hubiera quedado unificada a *normal*, por lo que la ejecución habría sido satisfactoria.

```
profiler/paquete_main([Ljava/lang/String;)I([l(0)],[s(1)],[e(1)],[A]) :=
 bytecode(-1, init_vars, [], [], [stack_types=t([], [])])
 bytecode(0, push(b, 6), [], [s(1)], [stack_types=t([], [primitiveType(byte)]), nextaddr=2])
 bytecode(2, push(i, 0), [], [s(2)], [stack_types=t([primitiveType(byte)], [primitiveType(int),
primitiveType(byte)]), nextaddr=3])
 nop(bytecode(3, barithm(i, div), [s(1), s(2)], [s(1)], [stack_types=t([primitiveType(int),
primitiveType(byte)], [primitiveType(int)]), nextaddr=4]), [consider_cost(false)])
 call(block, 5, [[l(0), s(1), s(2)], [s(1)], [e(1)], [A]], [])


5([l(0), s(1), s(2)],[s(1)],[e(1)],[A]) :=
 assr(int_nonzero, [s(2)], [])
 call(block, 3, [[l(0), s(1), s(2)], [s(1)], [e(1)], [A]], [])


5([l(0), s(1), s(2)],[s(1)],[e(1)],[A]) :=
 assr(int_zero, [s(2)], [])
 bytecode(-3, ie_throw(java/lang/ArithmeticException), [], [s(1)], [])
 call(block, 2, [[l(0), s(1)], [], [e(1)], [A]], [])


2([l(0), s(1)],[],[e(1)],[exception]) :=
 bytecode(-2, assignment, [s(1)], [e(1)], [stack_types=t([refType(classType(A))],
[refType(classType(B))])])


3([l(0), s(1), s(2)],[s(1)],[e(1)],[normal]) :=
 bytecode(3, barithm(i, div), [s(1), s(2)], [s(1)], [stack_types=t([primitiveType(int),
primitiveType(byte)], [primitiveType(int)]), nextaddr=4])
 bytecode(4, return(i), [s(1)], [s(1)], [stack_types=t([primitiveType(int)], [primitiveType(int)]),
nextaddr=5])
```

Para llevar a cabo este proceso de asignación y propagación en la implementación basta con aprovechar las características de las unificación Prolog y sus repercusiones y para que las guardas que fallen no afecten en esta asignación negativamente, se utiliza el backtraking.

La claúsula ***step_method_execution(*** *call(_Type, NewRuleHead,[InArgs, OutArgs, Exceptions, CallState],_)****,*** *RuleHead****)*** es la encargada de ejecutar el método. La variable 'CallState' lleva unificada la lista con todas las variables que configuran el estado actual. Una vez inicializado el

nuevo marco y unificada la cabecera de la regla, se llama a la claúsula ***stepRule(****NewRuleHead,CallState****)***, que se desglosa así:

```
stepRule(RuleHead,CallState) :-
   rbr_rule(RuleHead, [_, _, _, CallState], Bytecodes, _),
   stepBytecodes(Bytecodes, RuleHead),!.
```

'rbr_rule' unifica la cabecera de la regla, 'CallState' unifica con el estado y la lista 'Bytecodes' unifica con la lista que tiene los bytecodes de la regla.

stepBytecodes prosigue con la ejecución de los bytecodes de la regla. Si las guardas fallan, 'rbr_rule' obtendrá la siguiente regla y de nuevo 'CallState' se unificará con el nuevo estado. El resto del marco no importa que permanezca igual.

Conteo de instrucciones

Para proceder al conteo de instrucciones se utiliza el predicado dinámico *instructions_executed/1* y los predicados *init_instructions*, *increase_instructions* y *print_instructions_executed.*

En caso de haber indicado al intérprete que proceda al conteo de instrucciones el hecho *profile(instructions)* se encontrará en la base de hechos.

Durante la inicialización del intérprete se utiliza *init_instructions* para establecer a cero el contador de instrucciones asertando el siguiente hecho *instructions_executed(0)*.

A partir de aquí, cada predicado encargado del procesamiento de un bytecode utilizará el predicado *increase_instructions* que incrementará en uno el citado contador si se ha establecido el modo de conteo de instrucciones, ignorándose dicho comportamiento en caso contrario.

Cuando el intérprete llegue al final de la ejecución, utilizará el predicado *print_instructions_executed* para mostrar la información de las instrucciones ejecutadas al usuario.

Conteo de llamadas a métodos

Para proceder al conteo de llamadas a métodos se utiliza el predicado dinámico *methods_executed/2* y los predicados *increase_methods/1* y *print_methods_executed*.

El desglose de *methods_executed/2* es el siguiente: *methods_executed(nombre-metodo, n)* donde 'nombre-metodo' se corresponderá con el nombre del método definido en la RBR y 'n' se corresponderá con el número de veces que dicho método se ha ejecutado.

En caso de haber indicado al intérprete que proceda al conteo de llamadas a métodos el hecho *profile(methods, modo)* se encontrará en la base de hechos. 'modo' estará instanciado a *all* si se desea que se contabilicen todas las llamadas a todos los métodos o se corresponderá con el nombre de un método contenido en la RBR en caso de que solo desee contabilizarse ése en concreto.

A partir de aquí, cada vez que se produzca una llamada a un método y éste se corresponda con el método a contar o se haya especificado que se contabilicen todas las llamadas a métodos, se utilizará el predicado *increase_methods(nombre-metodo)* para contabilizar dicha llamada. 'nombre-metodo' se corresponderá con el nombre del método definido en la rbr del que se acaba de proceder a su llamada.

Cuando el intérprete llegue al final de la ejecución, utilizará el predicado *print_methods_executed* para mostrar la información de las llamadas a métodos producidas al usuario.

Conteo de objetos creados

Para proceder al conteo de objetos creados se utiliza el predicado dinámico *news_executed/2* y los predicados *increase_news/1* y *print_news_executed*.

El desglose de *news_executed/2* es el siguiente: *news_executed(tipo-objeto, n)* donde 'tipo-objeto' se corresponderá con el tipo del objeto creado y 'n' se corresponderá con el número de veces que dicho objeto se ha creado.

En caso de haber indicado al intérprete que proceda al conteo de objetos creados el hecho *profile(objects)* se encontrará en la base de hechos.

A partir de aquí, cada vez que se produzca la creación de un objeto, se utilizará el predicado *increase_new(tipo-objeto)* para contabilizar dicha creación. 'tipo-objeto' se corresponderá con el tipo del objeto que acaba de ser creado.

Cuando el intérprete llegue al final de la ejecución, utilizará el predicado *print_news_executed*' para mostrar la información de los objetos creados al usuario.

Ejemplos utilizados para la implementación

Los ejemplos utilizados durante la implementación del intérprete se encuentran en la carpeta /costa-extra/Test_Programs/profiler/, tienen extensión .java y prueban lo siguiente:

- *array1*: arrays unidimensionales de tipos simples integer, float, double, long y char.
- *array2*: arrays multidimensionales de tipos simples.
- *array3*: arrays unidimensionales de objetos.
- *array4*: arrays multidimensionales de objetos (Strings).
- *array5*: herencia y polimorfismo en arrays unidimensionales de objetos.
- *Ex1*, *Ex2*, *Ex3*, Ex4, *Ex5*: manejo de excepciones.
- *inargs1*: paso de array de String como parámetro de entrada.
- *inargs2*: paso de tipos simples integer, float, long, double y char y tipo String como parámetros de entrada.
- *profiler1*: operaciones básicas con tipos integer, long float y double.
- *profiler2*: estructuras de control for y while.
- *profiler3*: llamadas recursivas a bloques.
- *profiler4*, *profiler5*, *profiler6*: creación de objetos, acceso a atributos y herencia.

- *string1*: creación de un String y muestra de su contenido por pantalla.
- *string2*: comparación de Strings.
- *static1*, *static2*: acceso y manipulación de campos estáticos.
- *ret*: devolución de valores de retorno de tipos simples integer, float, long, double y char y de tipo String.

Anexo : Bytecodes de la máquina virtual de java

Habitualmente se muestran los bytecodes con una representación de instrucciones similar a un lenguaje ensamblador -en inglés se dice que es el *mnemoric* que corresponde a un bytecode- en él además se ven los operandos con los que trabaja. No se incluyen en esta sintaxis los operandos que toman de la pila, sólo las operaciones de tipo store y las de mutación de atributos dejan sus resultados en fuera de la pila, el resto, si lo dejan, lo hacen en la cima de la pila.

Es muy común encontrar en las instrucciones, los prefijos y sufijos *'i', 'l', 's', 'b', 'c', 'f', 'd', o 'a'*, que se refieren respectivamente a los *tipos, integer, long, short, byte, character, float, double*o *reference*, para indicar que estas hacen uso de operandos del tipo indicado.

Exceptuando los primeros códigos que mostramos, que están en desuso, siempre nos referiremos a los bytecodes utilizando su nombre *(mnemoric),* en las definiciones de lo que hace cada bytecode el operador *objectref,* proviene de la pila*, la llamada "constant pool"* en java, se ha traducido como *tabla de constantes.*

Códigos que no corresponden a instrucciones

El código *0x00* corresponde a nop y no hace nada, el código *0xBA* no se utiliza por "motivos históricos", *0xCA* sirve para poner puntos de parada cuando se usa un depurador. Los códigos *0xFE* y *0xFF* son utilizados internamente por la máquina virtual. Por último los códigos desde *0xCB* hasta*0xFD* han sido reservados para futuros usos.

Códigos de carga y descarga de operandos en la pila

Son del tipo Load, carga de constantes en la pila y Store:

- Load: Apilan un valor, pueden ser de los siguientes tipos:

aload index	Carga una referencia desde la variable local #index
aload_0	Carga una referencia desde la variable local 0
aload_1	Carga una referencia desde la variable local 1
aload_2	Carga una referencia desde la variable local 2
aload_3	Carga una referencia desde la variable local 3
baload	Carga un byte o un booleano value desde un vector
caload	Carga un char desde un vector
daload	Carga un double desde un vector
dload index	Carga un double desde la variable local #index
dload_0	Carga un double desde la variable local 0
dload_1	Carga un double desde la variable local 1
dload_2	Carga un double desde la variable local 2
dload_3	Carga un double desde la variable local 3
faload	Carga un float desde un vector
fload index	Carga un float desde la variable local #index

fload_0	Carga un float desde la variable local 0
fload_1	Carga un float desde la variable local 1
fload_2	Carga un float desde la variable local 2
fload_3	Carga un float desde la variable local 3
iaload	Carga un int desde un vector
iload index	Carga un int desde una variable #index
iload_0	Carga un entero desde la variable 0
iload_1	Carga un entero desde la variable 1
iload_2	Carga un entero desde la variable 2
iload_3	Carga un entero desde la variable 3
laload	Carga un long desde un vector
lload index	Carga un valor long desde la variable local #index
lload_0	Carga un valor long desde la variable local 0
lload_1	Carga un valor long desde la variable local 1
lload_2	Carga un valor long desde la variable local 2
lload_3	Carga un valor long desde la variable local 3
saload	Carga un short desde un array

- Carga de constantes: Adicionalmente hay instrucciones para cargar constantes:

aconst_null	apila una referencia a null en la pila
bipush byte	apila un byte en la pila como un entero
dconst_0	apila la constante 0.0 en la pila
dconst_1	apila la constante 1.0 en la pila
fconst_0	apila 0.0 de tipo flotante en la pila
fconst_1	apila 1.0 de tipo flotante en la pila
fconst_2	apila 2.0 de tipo flotante en la pila
lconst_0	apila 0 de tipo long en la pila
lconst_1	apila 1 de tipo long en la pila
ldc index	apila una constante #index desde la tabla de constantes (String, int o float) en la pila
ldc_w indexbyte1, indexbyte2	apila una constante #index desde la tabla de constantes (String, int o float) en la pila (el índice wide se calcula como indexbyte1 << 8 + indexbyte2)
ldc2_w indexbyte1, indexbyte2	apila una constante #index desde la tabla de constantes (double o long) en la pila (el índice wide se calcula como indexbyte1 << 8 + indexbyte2)
sipush byte1, byte2	apila un integer con signo (byte1 << 8 + byte2) en la pila

- Store: Cargan valores, en variables, que desapilan, pueden ser de los siguientes tipos:

aastore	Guarda una referencia en un array tomando de la pila el indice, la referencia al array y la referencia
astore index	Guarda una referencia en la variable local #index
astore_0	Guarda una referencia en la variable local 0
astore_1	Guarda una referencia en la variable local 1

astore_2	Guarda una referencia en la variable local 2
astore_3	Guarda una referencia en la variable local 3
bastore	Guarda un byte o booleano en un array tomando de la pila el indice, la referencia al array y el byte o booleano
castore	Guarda un char en un array tomando de la pila el índice, la referencia al array y el char
dastore	Guarda un double en un array tomando de la pila el indice, la referencia al array y el double
dstore index	Guarda un double value en la variable local #index
dstore_0	Guarda un double en la variable local 0
dstore_1	Guarda un double en la variable local 1
dstore_2	Guarda un double en la variable local 2
dstore_3	Guarda un double en la variable local 3
fastore	Guarda un float en un array tomando de la pila el índice, la referencia al array y el float
fstore index	Guarda un float en la variable local #index
fstore_0	Guarda un float en la variable local 0
fstore_1	Guarda un float en la variable local 1
fstore_2	Guarda un float en la variable local 2
fstore_3	Guarda un float en la variable local 3
iastore	Guarda un int en un array tomando de la pila el índice, la referencia al array y el int
istore index	Guarda un int en la variable #index
istore_0	Guarda un int en la variable 0
istore_1	Guarda un int en la variable 1
istore_2	Guarda un int en la variable 2
istore_3	Guarda un int en la variable 3
lastore	Guarda un long en un array tomando de la pila el índice, la referencia al array y el long
lstore index	Guarda un long en la variable local variable #index
lstore_0	Guarda un long en la variable local variable 0
lstore_1	Guarda un long en la variable local variable 1
lstore_2	Guarda un long en la variable local variable 2
lstore_3	Guarda un long en la variable local variable 3
sastore	Guarda un short en un array tomando de la pila el índice, la referencia al array y el short

Instrucciones Aritmético-lógicas

Son aquellas que llevan a cabo operaciones aritméticas o lógicas:

- Suma:

dadd	suma dos doubles
fadd	suma dos floats
iadd	suma dos ints together
ladd	suma dos longs
iinc index, const	Incrementa la variable local #index en un valor const, que debe ser de longitud byte con signo

- Resta:

dsub	resta un double a otro double
fsub	resta un float a otro float
isub	resta un int a otro int
lsub	resta un long a otro long

- Multiplicación:

dmul	multiplica dos doubles
fmul	multiplica dos floats
imul	multiplica dos integers
lmul	multiplica dos longs

- División:

ddiv	divide un double entre otro
fdiv	divide un float entre otro
idiv	divide un entero entre otro
ldiv	divide un long entre otro

- Resto / Módulo:

drem	calcula el resto de la división entre dos doubles
frem	calcula el resto de la división entre dos floats
irem	calcula el resto de la división entera entre dos ints
lrem	calcula el resto de la división entre entre dos longs

- Comparación y Desplazamiento de bits:

dcmpg	compara dos doubles
dcmpl	compara dos doubles (pero trata de diferente forma la comparación con NaN)
fcmpg	compara dos floats
fcmpl	compara dos floats (pero trata de diferente forma la comparación con NaN)
lcmp	compara dos longs
ishl	desplazamiento a la izquiera de enteros tantas posiciones como otro operando que toma de la pila
ishr	desplazamiento a la derecha de enteros tantas posiciones como otro operando que toma de la pila
iushr	desplazamiento lógico (trata igual el bit signo) a la derecha tantas posiciones como otro operando que toma de la pila
lshl	desplazamiento a la izquierda de un long tantas posiciones como otro operando que toma de la pila
lshr	desplazamiento a la derecha de un long tantas posiciones como otro operando que toma de la pila

lushr desplazamiento a la derecha de un long sin signo tantas posiciones como otro operando que toma de la pila

- Lógicas: And, Or, Xor, Not

iand Y lógica de 2 ints
land Y lógica de 2 longs
ior O lógica de 2 ints
lor O lógica de 2 longs
ixor O lógica exclusiva de 2 ints
lxor O lógica exclusiva de 2 longs
dneg niega bit a bit un double
fneg niega bit a bit un float
ineg niega bit a bit un int
lneg niega bit a bit un long

Tratamiento de tipos

- Conversion: Toman un valor de la pila de un tipo origen e introducen otro convertido al tipo destino en la misma:

d2f convierte un double en un float
d2i convierte un double en un int
d2l convierte un double en un long
f2d convierte un float en un double
f2i convierte un float en un int
f2l convierte un float en un long
i2b convierte un int en un byte
i2c convierte un int en un character
i2d convierte un int en un double
i2f convierte un int en un float
i2l convierte un int en un long
i2s convierte un int en un short
l2d convierte un long en un double
l2f convierte un long en un float
l2i convierte un long en un int

- Chequeo de tipos: Comprueba tipos

instanceof indexbyte1,indexbyte2 Determina si el objeto *objectref* es de in tipo dado, identificado por un índice en la tabla de constantes que se obtiene : (*indexbyte1 << 8 + indexbyte2*)

Gestión de objetos

- Creacion:

anewarray indexporte1, indexporte2 crea un nuevo array de referencias de tamaño length y de tipo type identificado por the una referencia a una clase index (indexporte1 << 8 + indexporte2) en la tabla de constantes

multianewarray indexporte1, indexporte2, dimensions	crea un nuevo array de dimension *dimensions* con elements de tipo identificado por una referencia a una clase en la tabla de constantes index (indexporte1 << 8 + indexporte2); el tamaño de cada dimension está determinado por count1, [count2, etc]
new indexporte1, indexporte2	crea un nuevo object de tipo identificado por una referencia a una clase en la tabla de constantes index (indexporte1 << 8 + indexporte2)
newarray atype	crea un nuevo array con count elementos de tipo primitivo identificado por atype

- Manipulación

getfield index1, index2	obtiene el valor de un atributo de un objeto, donde el atributo está identificado por una referencia a la tabla de constantes en el índice (index1 << 8 + index2)
getstatic index1, index2	obtiene el valor de un atributo estático de una clase, donde el atributo está identificado por una referencia a la tabla de constantes en el índice (index1 << 8 + index2)
putfield indexbyte1, indexbyte2	guarda el valor de un atributo en un objeto, donde el atributo está identificado por a una referencia a la tabla de constantes en el índice (indexbyte1 << 8 + indexbyte2)
putstatic indexbyte1, indexbyte2	guarda el valor de un atributo estático en una clase, donde el atributo está identificado por a una referencia a la tabla de constantes en el índice (indexbyte1 << 8 + indexbyte2)

Gestión de la pila

- Son operaciones que permiten duplicar, intercambiar o desapilar elementos innecesarios:

dup	apila la cima de nuevo en la pila, duplicándola
dup_x1	inserta una copia de la cima de la pila en el antepenúltimo lugar, dejando igual el último y penúltimo lugar
dup_x2	inserta una copia de la cima de la pila un nivel por dejajo del antepenúltimo lugar, dejando igual el último, penúltimo y antepenúltimo lugar
dup2	apila las dos últimas palabras (cima y anterior) de nuevo en la pila, duplicándolas, si es un tipo doble palabra, sólo se duplicará un valor
dup2_x1	inserta una copia de las dos últimas palabras de la pila en el antepenúltimo lugar, dejando igual el último y penúltimo lugar, si es un tipo doble palabra, sólo se duplicará un valor
dup2_x2	inserta una copia de las dos últimas palabras de la pila un nivel por debajo del antepenúltimo lugar, dejando igual el último, penúltimo y antepenúltimo lugar, si es un tipo doble palabra, sólo se duplicará un valor
swap	Intercambia las dos últimas palabras de la pila (por tanto long y double no se pueden usar)
pop	desapila una palabra de la cima de la pila
pop2	desapila dos valores de la cima o un valor si este es long o double

Control de flujo de ejecución

- Permiten modificar la secuencia lineal de código:

goto branchbyte1, branchbyte2	salto relativo desde la dirección actual a (signed short branchbyte1 << 8 + branchbyte2)
goto_w branchbyte1, branchbyte2, branchbyte3, branchbyte4	salto relativo desde la dirección actual a (signed int branchbyte1 << 24 + branchbyte2 << 16 + branchbyte3 << 8 + branchbyte4)
if_acmpeq branchbyte1, branchbyte2	si las referencias son iguales, salta a branchoffset (signed short construida desde branchbyte1 << 8 + branchbyte2)
if_acmpne branchbyte1, branchbyte2	si las referencias no son iguales, salta a la instrucción en branchoffset (signed short construida desde branchbyte1 << 8 + branchbyte2)
if_icmpeq branchbyte1, branchbyte2	si los enteros son iguales, salta a la instrucción en branchoffset (signed short construida desde branchbyte1 << 8 + branchbyte2)
if_icmpne branchbyte1, branchbyte2	si los enteros no son iguales salta a la instrucción en branchoffset (signed short construida desde branchbyte1 << 8 + branchbyte2)
if_icmplt branchbyte1, branchbyte2	si value1 es menor que value2, salta a la instrucción en branchoffset (signed short construida desde branchbyte1 << 8 + branchbyte2)
if_icmpge branchbyte1, branchbyte2	si value1 es mayor o igual que value2, salta a la instrucción en branchoffset (signed short construida desde branchbyte1 << 8 + branchbyte2)
if_icmpgt branchbyte1, branchbyte2	si value1 es mayor que value2, salta a la instrucción en branchoffset (signed short construida desde branchbyte1 << 8 + branchbyte2)
if_icmple branchbyte1, branchbyte2	si value1 es menor o igual que value2, salta a la instrucción en branchoffset (signed short construida desde branchbyte1 << 8 + branchbyte2)
ifeq branchbyte1, branchbyte2	si el valor es 0, salta a la instrucción en branchoffset (signed short construida desde branchbyte1 << 8 + branchbyte2)
ifne branchbyte1, branchbyte2	si el valor no es 0, salta a la instrucción en branchoffset (signed short construida desde branchbyte1 << 8 + branchbyte2)
iflt branchbyte1, branchbyte2	si el valor es menor que 0, salta a la instrucción en branchoffset (signed short construida desde branchbyte1 << 8 + branchbyte2)
ifge branchbyte1, branchbyte2	si el valor es mayor o igual que 0, salta a la instrucción en branchoffset (signed short construida desde branchbyte1 << 8 + branchbyte2)
ifgt branchbyte1, branchbyte2	si el valor es mayor que 0, salta a la instrucción en branchoffset (signed short construida desde branchbyte1 << 8 + branchbyte2)
ifle branchbyte1, branchbyte2	si el valor es menor o igual que to 0, salta a la instrucción en branchoffset (signed short construida desde branchbyte1 << 8 + branchbyte2)
ifnonnull branchbyte1, branchbyte2	si el valor es not null, salta a la instrucción en branchoffset (signed short construida desde branchbyte1 << 8 + branchbyte2)

ifnull branchbyte1, branchbyte2	si el valor es null, salta a la instrucción en branchoffset (signed short construida desde branchbyte1 << 8 + branchbyte2)
lookupswitch <0-3 bytes padding>, defaultbyte1, defaultbyte2, defaultbyte3, defaultbyte4, npairs1, npairs2, npairs3, npairs4, match-offset pairs...	se busca en la tabla una dirección destino utilizando una clave de búsqueda y la ejecución continúa desde la dirección que se obtiene
tableswitch [0-3 bytes padding], defaultbyte1, defaultbyte2, defaultbyte3, defaultbyte4, lowbyte1, lowbyte2, lowbyte3, lowbyte4, highbyte1, highbyte2, highbyte3, highbyte4, jump offsets...	Continúa la ejecución desde una dirección que se encuentre en la tabla con un offset index
jsr	salto a la subrutina que se encuentra en la dirección branchoffset (signed construida desde branchbyte1 << 8 + branchbyte2) y apila la dirección de retorno
Jsr_w	salto a la subrutina que se encuentra en la dirección branchoffset (signed int contruida a partir de la expresión sin signo *branchbyte1 << 24 + branchbyte2 << 16 + branchbyte3 << 8 + branchbyte4*) y apila la dirección de retorno

Invocación a métodos y retorno de los mismos

- Permiten cambiar de métodos y devolver valores, a diferencia de los saltos, aquí si se cambia el ámbito:

invoqueinterface indexbyte1, indexbyte2, count, 0	invoca un método de un interfaz que está en objectref, donde el método está identificado por una referencia en la tabla de constantes (indexbyte1 << 8 + indexbyte2)
invoquespecial indexbyte1, indexbyte2	invoca una instancia de un método que está en objectref, donde el método está identificado por una referencia en la tabla de constantes (indexbyte1 << 8 + indexbyte2)
invoquestatic indexbyte1, indexbyte2	invoca un método estático, donde el método está identificado por una referencia en la tabla de constantes (indexbyte1 << 8 + indexbyte2)
invoquevirtual indexbyte1, indexbyte2	invoca un método virtual method que está en objectref, donde el método está identificado por una referencia en la tabla de constantes (indexbyte1 << 8 + indexbyte2)
areturn	retorna una referencia desde un método
dreturn	retorna un double desde un método

freturn	retorna un float desde un método
ireturn	retorna un entero desde un método
lreturn	retorna un long desde un método
return	retorna vacío (void) desde un método void

ÍNDICE BIBLIOGRÁFICO

- ELVIRA ALBERT, PURI ARENAS and SAMIR GENAIM
  Cost Analysis of Object-Oriented Bytecode Programs
- ELVIRA ALBERT, PURI ARENAS, SAMIR GENAIM, GERMÁN PUEBLA and DAMIANO ZANARDINI
  Resource Usage Analysis and Its Application to Resource Certification
- TIM LINDHOLM and FRANK YELLIN
  The Java™ Virtual Machine Specification, Second Edition
  http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html
- WIKIPEDIA
  Java (programming language), History.
  http://en.wikipedia.org/wiki/Java_(programming_language)

  Java virtual machine, JVM languages.
  http://en.wikipedia.org/wiki/Java_virtual_machine

Los autores de este proyecto de Sistemas Informáticos autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Firmado:

Carlos Loredo Iglesias Sergio Ortiz Gil Héctor Valles Mercado